Wait-Free Updates and Range Search using Uruv

Gaurav Bhardwaj[1], Bapi Chatterjee[2], Abhay Jain[1], and Sathya Peri[1]

[1] Indian Institute of Technology Hyderabad, Hyderabad, India
[2] Indraprastha Institute of Information Technology Delhi, Delhi, India

Abstract. CRUD operations, along with range queries make a highly useful abstract data type (ADT), employed by many dynamic analytics tasks. Despite its wide applications, to our knowledge, no fully wait-free data structure is known to support this ADT. In this paper, we introduce Uruv, a proactive linearizable and practical wait-free concurrent data structure that implements the ADT mentioned above. Structurally, Uruv installs a balanced search index on the nodes of a linked list. Uruv is the first wait-free and proactive solution for concurrent B^{+}tree. Experiments show that Uruv significantly outperforms previously proposed lock-free B^{+}trees for dictionary operations and a recently proposed lock-free method to implement the ADT mentioned above.

Keywords: Wait-Free · Lock-Free · Range Search · B+ Tree

1 Introduction

With the growing size of main memory, the in-memory big-data analytics engines are becoming increasingly popular [25]. Often the analytics tasks are based on retrieving keys from a dataset specified by a given range. Additionally, such applications are deployed in a streaming setting, e.g., Flurry [12], where the dataset ingests real-time updates. Ensuring progress to every update would be attractive for many applications in this setting, such as financial analytics [21]. The demand for real-time high-valued analytics, the powerful multicore CPUs, and the availability of large main memory together motivate designing scalable concurrent data structures to utilize parallel resources efficiently.

It is an ever desirable goal to achieve *maximum progress* of the concurrent operations on a data structure. The maximum progress guarantee – called *wait-freedom* [14] – ensures that each concurrent non-faulty thread completes its operation in a finite number of steps. Traditionally, wait-freedom has been known for its high implementation cost and subsided performance. Concomitantly, a weaker guarantee that some non-faulty threads will finitely complete their operations – known as *lock-freedom* – has been a more popular approach. However, it has been found that the lock-free data structures can be transformed to *practical* wait-free [16] ones with some additional implementation and performance overhead. Progress promises of wait-free data structures make their development

imperative, to which a practical approach is to co-design them with their efficient lock-free counterpart. While a progress guarantee is desirable, consistency of concurrent operations is a necessity. The most popular consistency framework is *linearizability* [15], i.e., every concurrent operation emerges taking effect at an atomic step between its invocation and return.

In the existing literature, the lock-free data structures such as k-ary search trees [7], and the lock-based key-value map KiWi [3] provide range search. In addition, several generic methods of concurrent range search have been proposed. Chatterjee [9] presented a lock-free range search algorithm for lock-free linked-lists, skip-lists, and binary search trees. Arbel-Raviv and Brown [1] proposed a more generic approach associated with memory reclamation that fits into different concurrency paradigms, including lock-based and software transactional memory-based data structures. Recently, two more approaches – bundled-reference [19] and constant time snapshots [24] – were proposed along the same lines of generic design. Both these works derive from similar ideas of expanding the data structure with versioned updates to ensure linearizability of scans. While the former stores pointers with time-stamped updates, the latter adds objects to nodes time-stamped by every new range search. Moreover, bundled-reference [19] design requires locks in every node.

In most cases, for example [3,9,19,24], the range scans are unobstructed even if a concurrent modification (addition, deletion, or update) to the data structure starves to take even the first atomic step over a shared node or pointer. A reader would perceive, indeed for good reasons, that once the modifications are made wait-free the entire data structure will become wait-free. However, to our knowledge, none of these works actually investigates how trivial or non-trivial it would be to arrive at the final implementation of concurrent wait-free CRUD and range-search. This is exactly where our work contributes.

Proposed wait-free linearizable proactive data structure

In principle, Uruv's design derives from that of a B^+Tree [10], a self-balancing data structure. However, we need to make the following considerations:

Wait-freedom: Firstly, to ensure wait-freedom to an operation that needs to perform at least one `CAS` execution on a shared-memory word, it must *announce* its invocation [16]. Even if delayed, the announcement has to happen on realizing that the first `CAS` was attempted a sufficient number of times, and yet it starved [16]. The announcement of invocation is then followed by a *guaranteed help* by a concurrent operation at some finite point [16].

Linearizability: Now, to ensure linearizability of a scan requires that its output reflects the relevant changes made by every update during its lifetime. The technique of repeated multi-scan followed by validation [7], and collecting the updates at an augmented object, such as RangeCollector in [9], to let the range search incorporate them before it returns, have been found scaling poorly [7, 9]. Differently, multi-versioning of objects, for example [19], can have a (theoretical) possibility to stockpile an infinite number of versioned pointers between two nodes. Interestingly, [1] exploits the memory reclamation mechanism to synchro-

nize the range scans with delete operations via logically deleted nodes. However, for lock-freedom, they use a *composite primitive* double-compare-single-swap (DCSS). In comparison, [24] uses only single-word `CAS`. However, managing the announcement by a starving updater that performs the first `CAS` to introduce a versioned node to the data structure requires care for a wait-free design.

Node Structure: The "fat" (array-based) data nodes, for example Kiwi [3], improve traversal performance by memory contiguity [17]. However, the benchmarks in [24] indicate that it does not necessarily help as the number of concurrent updates picks up. Similarly, the lock-free B+trees by Braginsky and Petrank [5] used memory chunks, and our experiments show that their method substantially underperforms. Notwithstanding, it is wise to exploit memory contiguity wherever there could be a scope of "slow" updates in a concurrent setting.

Proactive maintenance: Finally, if the number of keys in a node exceeds (falls short of) its maximum (minimum) threshold after an insertion (deletion), it requires splitting (merging). The operation splitting the node divides it into two while adding a key to its parent node. It is possible that the split can percolate to the root of the data structure if the successive parent nodes reach their respective thresholds. Similarly, merging children nodes can cause cascading merges of successive parent nodes. With concurrency, it becomes extremely costly to tackle such cascaded split or merge of nodes from a leaf to the root. An alternative to this is a *proactive approach* which checks threshold of nodes whiles traversing down a tree every time; if a node is found to have reached its threshold, without waiting for its children, a pre-emptive split or merge is performed. As a result, a restructure remains localized. To our knowledge, no existing concurrent tree structure employs this proactive strategy.

With these considerations, we introduce a key-value store **Uruv** (or, Uruvriksha [c]) for wait-free updates and range search. More specifically,

(a) Uruv stores keys with associated values in leaf nodes structured as linked-list. The interconnected leaf nodes are indexed by a balanced tree of fat nodes, essentially, a classical B+ Tree [10], to facilitate fast key queries. (Section 2)

(b) The key-nodes are augmented with list of versioned nodes to facilitate range scans synchronize with updates. (Section 3)

(c) Uruv uses single-word `CAS` primitives. Following the fast-path-slow-path technique of Kogan and Petrank [16], we *optimize* the helping procedure for wait-freedom. (Section 4). We prove linearizability and wait-freedom and present the upper bound of step complexity of operations. (Section 5)

(d) Our C++ implementation of Uruv significantly outperforms existing similar approaches – lock-free B+tree of [5], and OpenBWTree [23] for dictionary operations. It also outperforms a recently proposed method by Wei et al. [24] for concurrent workloads involving range search. (Section 6)

A full version of the paper is available at `http://arxiv.org/abs/2307.14744` [4]. Please refer to the full version for detailed pseudocode.

---

[c] Uruvriksha is the Sanskrit word for a wide tree.

2 Preliminaries

We consider the standard shared-memory model with atomic `read`, `write`, `FAA` (fetch-and-increment), and `CAS` (compare-and-swap) instructions. Uruv implements a key-value store $(\mathcal{K}, \mathcal{V})$ of keys $K \in \mathcal{K}$ and their associated values $V \in \mathcal{V}$.

The Abstract Data Type (ADT): We consider an ADT $\mathcal{A}$ as a set of operations: $\mathcal{A} = \{\textsc{Insert}(K, V),\ \textsc{Delete}(K),\ \textsc{Search}(K),\ \textsc{RangeQuery}(K1, K2)\}$

1. An $\textsc{Insert}(K, V)$ inserts the key K and an associated value V if $K \notin \mathcal{K}$.
2. A $\textsc{Delete}(K)$ deletes the key K and its associated value if $K \in \mathcal{K}$.
3. A $\textsc{Search}(K)$ returns the associated value of key K if $K \in \mathcal{K}$; otherwise, it returns -1. It does not modify $(\mathcal{K}, \mathcal{V})$.
4. A $\textsc{RangeQuery}(K_1, K_2)$ returns keys $\{K \in \mathcal{K} : K_1 \le K \le K_2\}$, and associated values without modifying $(\mathcal{K}, \mathcal{V})$; if no such key exists, it returns -1.

2.1 Basics of Uruv's Lock-free Linearizable Design

Uruv derives from a B$^+$Tree [10], a self-balancing data structure. However, to support linearizable range search operations, they are equipped with additional components. The key-value pairs in Uruv are stored in the *key nodes*. A *leaf node* of Uruv is a sorted linked-list of key nodes. Thus, the leaf nodes of Uruv differ from the array-based leaf nodes of a B$^+$Tree. The *internal nodes* are implemented by arrays containing ordered set of keys and pointers to its descendant children, which facilitate traversal from the root to key nodes. A search path in Uruv is shown in Figure 1.

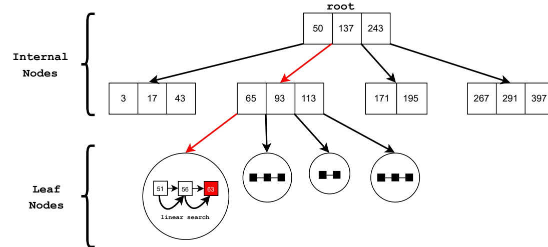

Fig. 1: Example of Uruv's design. In this example, a search operation is being performed wherein the red arrows indicate a traversal down Uruv, and we find the key, highlighted red, in the linked-list via a linear search.

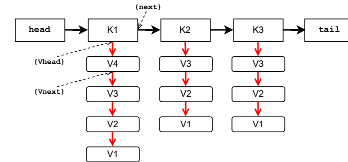

Fig. 2: Versioned key nodes

Unlike an array in a B+Tree's leaf node, the key nodes making leaf nodes of Uruv contain lists of versioning nodes mainly to ensure linearizability of range search operations. The mechanism of linearizable range search derives from that of Wei et al. [24] – every range search increments a data structure-wide version counter whereby concurrent addition and removal operations determine their versions. A range search returns the keys and corresponding values only if its

version is at most the version of the range search. Thus the linearization point of range search coincides with the atomic increment of version counter.

The associated values to a key are stored in the versioning nodes, see Figure 2. The creation of a key-value pair, its deletion, and addition back to the key-value store updates the version list with a version and associated value. With this design, Uruv supports concurrent linearizable implementation of the ADT operations as described above.

3 Lock-Free Algorithm

3.1 The structures of the component nodes

Here we first describe the structure of the nodes in Uruv. See Figure 3. A versioning node is implemented by the objects of type Vnode. A key node as described in the last section, is implemented by the objects of the class llNode. Nodes of type llNode make the linked-list of a leaf-node which is implemented by the class VLF_LL.

The leaf and internal nodes of Uruv inherit the Node class. See Figure 4. An object of class Node of Uruv, hereafter referred to as a node object, keeps count of the number of keys. A node object also stores a boolean to indicate if it is a leaf node. A boolean variable 'frozen' helps with "freezing a node" while undergoing a split or merge in a lock-free updatable setting. A thread on finding that a node is frozen helps the operation that triggered the freezing.

```
Vnode{                     llNode{                  VLF_LL{
    value_t value;              key_t key;
    int ts;                     Vnode* vhead;            llNode* head;
    Vnode* nextv;               llNode* next;
}                          }                        }
```

Fig. 3: Versioned Lock-Free Linked-List Data Structure

Every leaf node has three pointers *next*, *newNext* and a pointer to version list *ver_head* and one variable *ts* for the timestamp. The *next* pointer points to the next adjacent leaf node in Uruv. When a leaf node is split or merged, the *newNext* pointer ensures leaf connection. A new leaf node is created to replace it when a leaf node is balanced. Using the *newNext* pointer, we connect the old and new leaf nodes. When traversing the leaf nodes for RangeQuery with *newNext* set, we follow *newNext* instead of *next* since that node has been replaced by a newer node, ensuring correct traversal. The initial *ts* value is associated with the construction of the leaf node.

3.2 Versioned Linked-List

The description of lock-free linearizable implementation of the ADT operations RangeQuery, Insert, and Delete requires detailing the versioned linked list. A versioned list holds the values associated with the key held at various periods. Each versioned node (`Vnode`) in the versioned list has a value, the time

```
Uruv{
    Node* root;
}
InternalNode: Node{
    long key[MAX]
    Node* ptr[MAX+1]
    helpidx
}

LeafNode: Node{
    VLF_LL* ver_head;
    LeafNode* next;
    LeafNode* newNext;
    int ts
}

Node{
    long count;
    bool isLeaf;
    bool frozen;
}
```

Fig. 4: The details of object structures

when the value was modified, and a link to the previous version of that key. Versioned linked-list information may be seen in Figure 2 and Figure 3. The versioned list's nodes are ordered in descending order by the time they have been updated. Compared to the [13], there is no actual delinking of nodes; instead, we utilise a tombstone value (a special value not associated with any key) to indicate a deleted node. Moreover, deleting a node requires no help since there is no delinking. Although, for memory reclamation, we retain a record of active RangeQuery and release nodes that are no longer needed. Any modification to the versioned linked list atomically adds a version node to the `vhead` of `llNode` using `CAS`.

3.3 Traversal and Proactive maintenance in Uruv

We traverse from root to leaf following the order provided by the keys in the internal nodes. In each internal node, a binary search is performed to determine the appropriate child pointer. While traversal in Insert and Delete operations, we follow the proactive approach as described earlier. Essentially, if we notice that a node's key count has violated the maximum/minimum threshold, we instantly conduct a split/merge action, and the traversal is restarted. The proactive maintenance is shown in Figure5.

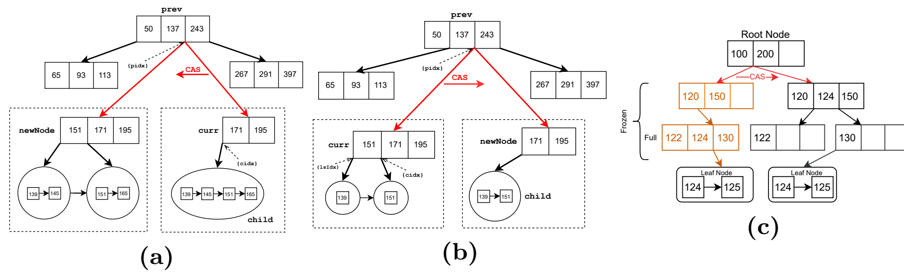

Fig. 5: (a) Split Leaf, (b) Merge Leaf, (c) Split Internal

3.4 ADT Operations

```
1:  Insert(key, value)
2:    retry:
3:    Node* curr := root
4:    if curr = nullptr then
5:      Node* nLeaf →insLeaf(key, value)
6:      if !root.CAS(curr, nLeaf) then
7:        goto retry
8:      else return
9:    curr := balanceRoot(curr)
10:   if !curr then goto retry
11:   Node* prev, child := nullptr
12:   int pidx, cidx
13:   while !curr → isLeaf do
14:     if curr → helpIdx ≠ −1 then
15:       Node* res := help(prev, pidx, curr)
16:       if res then curr := res
17:       else goto retry
18:     cidx is set to the index of appropriate
        child based on key using Binary Search
19:     child := curr → ptr[cidx]
20:     if child → isLeaf && child →
        frozen then
21:       curr → freezeInternal()
22:       if !curr →setHelpIdx(cidx) then
23:         goto retry
24:       Node*newNode := child →
          balanceLeaf(prev, pidx, curr, cidx)
25:       if newNode then
26:         then curr := newNode
27:       else goto retry
28:     else if !child → isLeaf&& child →
        count ≥ MAX then
29:       curr → freezeInternal()
30:       if !curr →setHelpIdx(cidx) then
31:         goto retry
32:     Node* newNode := child →
        splitInternal(prev, pidx, curr, cidx)
33:     if newNode then
34:       then curr := newNode
35:     else goto retry
36:     prev := curr
37:     curr := child
38:     pidx := cidx
39:   res := curr → insertLeaf(key, value)
40:   if res = Failed then
41:     goto retry
42:   else return res
```

Fig. 6: Pseudocode of INSERT operation

An Insert operation starts with performing a traversal as described above to locate the leaf node to insert a key and its associated value. It begins with the root node; if it does not exist, it builds a new leaf node and makes it the root with a CAS. If it cannot update the root, another thread has already changed it, and it retries insertion. Method balanceRoot splits the root if needed and replaces it with a new root using CAS. If CAS fails, then some other thread must have changed the root, and it returns null. If there is no need to split the root, it will return the current root.

Lines 14-17 describe the helping mechanism, which makes the data structure lock-free. If any node helpIdx is set to a value other than -1, then the child node at helpIdx is undergoing the split/merge process. In that case, it will help that child finish its split/merge operation. Method `help` helps *child* node in split/merge operation and returns the new *curr* node if it successfully replaces it using CAS; otherwise, it returns null. Then, it performs a binary search over *curr*'s keys at line 18 to find the correct child pointer. It copies the child pointer into *child* and stores its index in the pointer array as *cidx*.

If the child node is a frozen leaf node or an internal node that has reached the threshold, it performs a split/merge operation. It starts by freezing its parent, *curr*, at line 21 by setting a special freezing marker on every child pointer, so that no other thread can change the parent node and cause inconsistency. After freezing the parent, it stores the index of the child pointer in helpIdx of the parent node using CAS so that other threads can help in split/merge operation. If setHelpIdx fails, that means some other thread has already set the helpIdx, and it retries.

Restructuring a *child* is performed at line 24 and 32 using balanceLeaf and splitInternal respectively. balanceLeaf performs the split/merge operation on the leaf node based on the number of elements and returns the node replaced by the parent node using CAS. Similarly, splitInternal splits the internal node and returns the new parent node. If in any of the above methods, CAS is failed, then some other thread must have replaced it, and it will return nullptr and retries at line 31 and 35. It repeats the same process until it reaches the leaf node. Once it reaches the leaf node, it performs the insert operation in the leaf node at line 39. It returns on success, otherwise it retries.

Insert into a leaf. In the leaf node, all the updates occur concurrently in the versioned linked list. It first checks if the leaf node is frozen. If it is, it returns "Failed", realizing that another thread is trying to balance this node. If the node's count has reached the maximum threshold, it freezes it and returns "Failed". Leaf node is frozen by setting a special freezing mark on llnode *next* pointer and the *vhead* pointer. In both the cases, when it returns "Failed" insertion will be retried after balancing it. Otherwise, it would insert the key into the versioned lock-free linked list. If another thread is concurrently freezing the leaf node, the insertion into the linked list might fail. If it fails, it will again return "Failed" and retries the insertion. If the key is already present in the linked list, it updates that key's version by adding a new version node in the version list head with a new value. Else it will create a new node in the linked list containing the key and its value in the version node. After the key is inserted/updated in the linked list, its timestamp is set to the current timestamp, which is the linearization point for insertion in the tree.

A Delete operation follows a similar approach as INSERT. It traverses the tree to the leaf node, where the key is present. The difference in traversal with respect to INSERT operation is that at line 28, instead of checking the max threshold, it checks for the minimum threshold. Instead of splitting the internal node at line 32, it merges the internal node. Once a leaf node is found, it checks whether the key is in the linked list. If it is in the linked list, it will update a tombstone value in the version list to mark that key as deleted. If the key is absent, it returns "Key not Present".

Delete from Leaf. If the key is present, this operation creates a versioned node with a tombstone value to set it as deleted. Just like inserting the new versioned node its timestamp is set to the current timestamp. If the key is not present in the linked list it simply returns "Key Not Present".

Search operation. Traversal to a leaf node in case of searching doesn't need to perform any balancing. After finding the leaf node, it checks the key in the linked list; if it is present, it returns the value from the version node from the head of the list; otherwise, it simply returns "Key not Present". Before reading the value from the versioned node it checks if the timestamp is set or not. If it is not set, it sets the timestamp as the current timestamp before reading the value.

RangeQuery. A range query returns keys and their associated values by a given range from the data. Uruv supports a linearizable range query employing a multi-version linked list augmented to the nodes containing keys. This approach

draws from Wei et al. [24]'s work. A global timestamp is read and updated every time a range query is run. The leaf node having a key larger than or equal to the beginning of the supplied range is searched after reading the current time. Then, it chooses a value for the relevant key from the versioned list of values. Figure 5(c) depicts a versioned linked list, with the higher versions representing the most recent modifications.

By iterating over each versioned node individually, it selects the first value in the list whose timestamp is smaller than the current one. This means that the value was changed before the start of the range query, making it consistent. It continues to add all keys and values that are less than or equal to the end of the given range. Because all of the leaf nodes are connected, traversing them is quick. After gathering the relevant keys and values, the range query will produce the result.

As a leaf node could be under split or merge, for every leaf node that we traverse, we first check whether their $newNext$ is set. If it is and the leaf pointed to by $newNext$ has a timestamp lower than the range query's timestamp, it traverses the $newNext$ pointer. This ensures that our range query collects data from the correct leaf nodes. Were the timestamp not part of the leaf node, there is a chance that the range query traverses $newNext$ pointers indefinitely due to repeated balancing of the leaf nodes.

4 Wait-Free Construction

We now discuss a wait-free extension to the presented lock-free algorithm above. Wait-freedom is achieved using fast-path-slow-path method [22]. More specifically, a wait-free operation starts exactly as the lock-free algorithm. This is termed as the *fast path*. If a thread cannot complete its operation even after several attempts, it enters the *slow path* by announcing that it would need help. To that effect, we maintain a global `stateArray` to keep track of the operations that every thread currently needs help with. In the slow path, an operation first publishes a `State` object containing all the information required to help complete its operation.

For every thread that announces its entry to the slow path, it needs to find helpers. After completing some fixed number of fast path operations, every thread will check if another thread needs some help. This is done by keeping track of the thread to be helped in a thread-local `HelpRecord` object presented in Figure 7. After completing the $nextCheck$ amount of fast path operations, it will assist the $currTid$. Before helping, it checks if $lastPhase$ equals $phase$ in $currTid$'s `stateArray` entry. If it does, the fast path thread will help execute the wait-free implementation of that operation; otherwise, $currTid$ doesn't require helping as its entry in the `stateArray` has changed, meaning the operation has already been completed. In the worst case, if the helping thread also faces massive contention, every available thread will eventually execute the same operation, ensuring its success.

Notice that when data and updates are uniformly distributed, the contention among threads is low, often none. Concomitantly, in such cases, a slow path by any thread is minimally taken.

```
State* stateArray[totalThreads]

class HelpRecord{
    long currTid;
    long lastPhase;
    long nextCheck;
}
```

```
class State{
    long phase;
    bool finished;
    Vnode* vnode;
    long key;
    long value;
    llNode* searchNode
}
```

Fig. 7: Data structures used in wait-free helping

Wait-free Insert. Traversal in Wait-free INSERT is the same as that in the lock-free INSERT as mentioned in Section 3. While traversal a thread could fail the `CAS` operation in a split/merge operation of a node and would need to restart traversal from the root again. At first glance, this would appear to repeat indefinitely, contradicting wait-freedom, but this operation will eventually finish due to helping. If a thread repeatedly fails to traverse Uruv due to such failure, every other thread will eventually help it find the leaf node. Once we reach the leaf node, we add the key to the versioned linked list as described below. There are two cases - either a node containing the key already exists, or a node does not exist.

In the former case, we need to update the linked list node's *vhead* with the versioned node, *vnode*, containing the new value using `CAS`. The significant difference between both methods is the usage of a shared `Vnode` from the `stateArray` in wait-free versus a thread local `Vnode` in lock-free. Every thread helping this insert will take this *vnode* from the `stateArray` and first checks the variable finished if the operation has already finished. They then check if the phase is the same in the `stateArray`, and *vnode*'s timestamp is set or not. If either is not true, some other thread has already completed the operation, and they mark the operation as finished. Else, they will try to update the *vhead* with *vnode* atomically. After inserting the *vnode*, it initialises the timestamp and sets the finished to be true.

In the latter case, we create a linked list node, *newNode*, and set its *vhead* to the *vnode* in the `stateArray` entry. It tries adding *newNode* like the lock-free linked list's insert. If it is successful, the timestamp of *vnode* is initialized, and the finished is set to true in the `stateArray`.

Wait-free Delete. DELETE operation follows the same approach as INSERT. If the key is not present in the leaf node, it returns "Key Not Present" and sets the finished to be true. Otherwise, it will add the *vnode* from `stateArray` similar to wait-free INSERT. The only difference is that the *vnode* contains the tombstone value for a deleted node.

Search and RangeQuery. Neither operation modifies Uruv nor helps any other operation; hence their working remain as explained in Section 3.

5 Correctness and Progress Arguments

To prove the correctness of Uruv, we have shown that Uruv is linearizable by describing *linearization points* (LPs) that map any concurrent setting to a sequential order of said operations. We discuss them in detail below.

5.1 Linearization Points

As explained earlier, we traverse down Uruv to the correct leaf node and perform all operations on the linked list in that leaf. Therefore, we discuss the LPs of the versioned linked list.

Insert*:* There are two cases. If the key does not exist, we insert the key into the linked list. However, the timestamp of the `vnode` is not set, so the `LP` for INSERT operation is when the timestamp of `vnode` is set to the current timestamp. This can be executed either just after the insertion of the key in the linked list or by some other thread before reading the value from `vnode`.

If the key already exists, we update its value by atomically replacing a new versioned node by its current *vhead*. After successfully changing the *vhead*, the node's timestamp is still not set. It can be set just after adding the new versioned node or by some other thread before reading the value from the newly added versioned node. In both the cases the `LP` is when the timestamp of the versioned node is set to the current timestamp.

Delete*.* There are two cases. If the key does not exist, then there is no need to delete the key as it does not exist. Therefore, the `LP` would be where we last read a node from the linked list. Instead, if the key exists, the `LP` will be same as INSERT when we set the timestamp of the versioned node.

Search*.* There are two cases, first if the key doesn't exist in the linked list, the SEARCH `LP` would be when we first read the node whose key is greater than the key we are searching for in the linked list. Second, if the key is present in the linked list it reads the value in the versioned node at `vhead`. So the `LP` is when we atomically reads the value from the versioned node. If a concurrent insert/delete leads to a split/merge operation, then there is a chance that the search will end up at a leaf node that is no longer a part of Uruv. In that case, the search's `LP` would have happened before insert/delete's `LP`. Search's `LP` remains the same as above.

RangeQuery*.* RANGEQUERY method reads the global timestamp and increment it by 1. So the `LP` for range query would be the atomic read of global timestamp. The range query's `LP` will remain the same regardless of any other concurrent operation.

6 Experiments

In this section, we benchmark Uruv against (a) previous lock-free variants of the B^+Tree for updates and search operations (to our knowledge, there are no existing wait-free implementations of the B^+Tree, and lock-free B^+Trees do not implement range search), and (b) the lock-free VCAS-BST of [24], which is the best-performing data structure in their benchmark. The code of the benchmarks is available at `https://github.com/PDCRL/Uruv.git`.

Experimental Setup. We conducted our experiments on a system with an IBM Power9 model 2.3 CPU packing 40 cores with a minimum clock speed of 2.30 GHz and a maximum clock speed of 3.8 GHz. There are four logical threads for each core, and each has a private 32KB L1 data cache and L1 instruction cache. Every pair of cores shares a 512KB L2 cache and a 10MB L3 cache. The system has 240GB RAM and a 2TB hard disk. The machine runs Ubuntu 18.04.6 LTS. We implement Uruv in C++. Our code was compiled using g++ 11.1.0 with -std=c++17 and linked the pthread and atomic libraries. We take the average of the last seven runs out of 10 total runs, pre-warming the cache the first three times. Our average excludes outliers by considering results closest to the median.

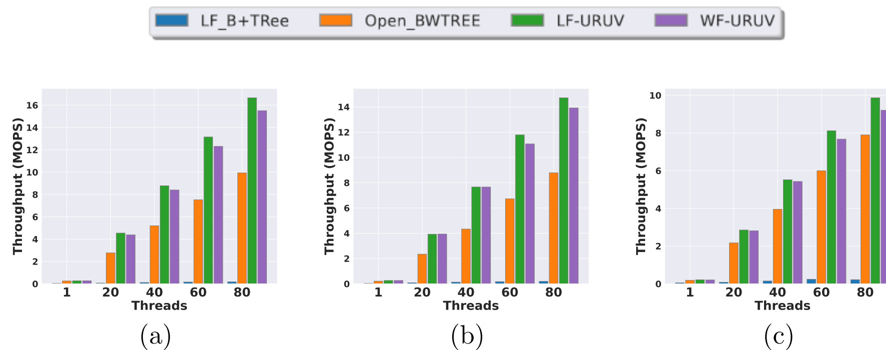

Fig. 8: The performance of **Uruv** when compared to **LF_B+Tree** [5] and **Open_BwTree** [23]. Higher is better. The workload distributions are (a) Reads - 100% (b) Reads - 95%, Updates - 5%, and (c) Reads - 50%, Updates - 50%

Benchmark. Our benchmark takes 7 parameters - read, insert, delete, range query, range query size, prefilling size, and dataset size. Read, insert, delete, and range queries indicate the percentage of these operations. We use a uniform distribution to choose between these four operations probabilistically. We prefill each data structure with 100 million keys, uniformly at random, from a universe of 500 million keys ranging [1, 500M].

Performance for dictionary operations. Results of three different workloads - Read-only(Fig. 8a), Read-Heavy(Fig. 8b), and a Balanced workload(Fig. 8c) are shown in Figure 8. Across the workloads, at 80 threads, Uruv beats LF_B$^+$Tree [5] by **95x**, **76x**, and **44x** as it replaces the node with a new node for every insert. Uruv beats OpenBwTree [23] by **1.7x**, **1.7x**, and **1.25x**. The performance of LF-URUV and WF-URUV correlates since WF-URUV has a lower possibility of any thread taking a slow path. In all three cases, the gap between Uruv and the rest increases as the number of threads increases. This shows the scalability of the proposed method. As we move from 1 to 80 threads, Uruv scales **46x** to **61x** in performance, LFB$^+$Tree scales **2.4x** to **5x** and OpenBw-Tree scales **39x** to **42x**. These results establish the significantly superior performance of Uruv over its existing counterpart.

Performance for workloads including range search. We compare Uruv against VCAS-BST in various workloads in Figure 9. Figures 9a - 9c are read-heavy workloads and 9d - 9f are update-heavy workloads. Across each type of workload, we vary the range query percentage from 1% to 10%. At 80 threads, we beat VCAS-BST by **1.38x** in update-heavy workloads and **1.68x** in read-heavy workloads. These set of results demonstrate the efficacy of Uruv's wait-free range search.

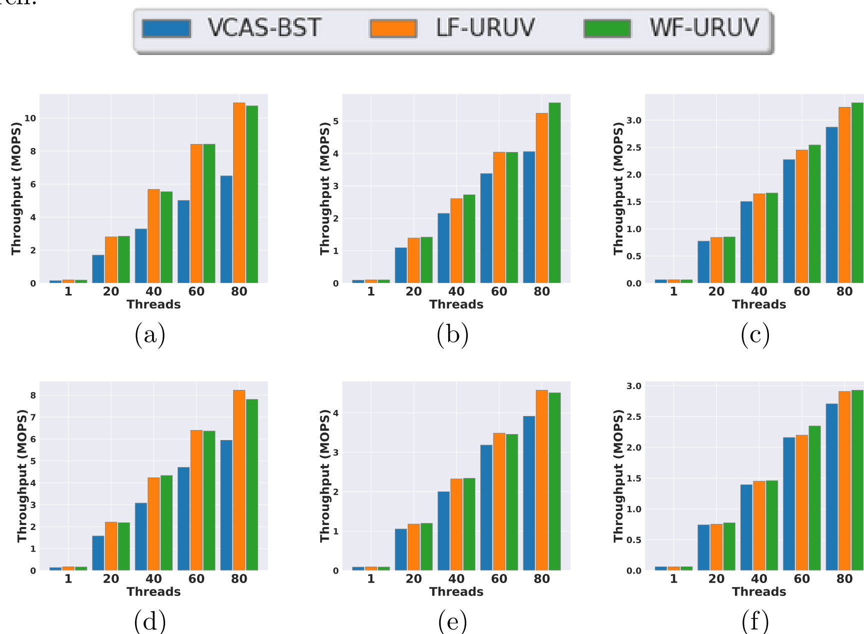

Fig. 9: The performance of **Uruv** when compared to **VCAS-BST**. The workload distributions are (a) Reads - 94%, Updates - 5%, Range Queries of size 1K - 1%, (b) Reads - 90%, Updates - 5%, Range Queries of size 1K - 5%, (c) Reads - 85%, Updates - 5%, Range Queries of size 1K - 10%, (d) Reads - 49%, Updates - 50%, Range Queries of size 1K - 1%, (e) Reads - 45%, Updates - 50%, Range Queries of size 1K - 5%, and (f) Reads - 40%, Updates - 50%, Range Queries of size 1K - 10%

7 Related Work

We have already discussed the salient points where Uruv differs from existing techniques of concurrent range search. In particular, in contrast to the locking method of bundled references [19] and the lock-free method of constant time snapshots [24], Uruv guarantees wait-freedom. The architecture ensuring wait-freedom in Uruv, i.e., its `stateArray`, has to accommodate its multi-versioning. The existing methods did not have to consider this.

Anastasia et al. [5] developed the first lock-free B$^+$Tree. In their design, every node implements a linked-list augmented with an array. This ensures that each node in the linked-list is allocated contiguously. It slows down updates at the leaf and traversal down their tree. Uruv's design is inspired by their work, but, does away with the arrays in the nodes. As the experiments showed, it clearly benefits. Most importantly, we also support linearizable wait-free range search, which is

not available in [5]. OpenBw-Tree [23] is an optimized lock-free B$^+$tree that was designed to achieve high performance under realistic workloads. However, again, it does not support range search.
We acknowledge that other recently proposed tree data structures could be faster than Uruv, for example, C-IST [8] and LF-ABTree [6]. However, LF-ABTree is a relaxed tree where the height and the size of the nodes are relaxed whereas C-IST [8] uses interpolation search on internal nodes to achieve high performance. That is definitely an attractive dimension towards which we plan to adapt the design of Uruv. Furthermore, they are not wait-free. Our focus was on designing a B$^+$Tree that supports wait-free updates and range search operations.
In regards to wait-free data structures, most of the attempts so far has been for Set or dictionary abstract data types wherein only insertion, deletion, and membership queries are considered. For example, Natarajan et al. [18] presented wait-free red-black trees. Applying techniques similar to fast-path-slow-path, which we used, Petrank and Timmet [20] proposed converting lock-free data structures to wait-free ones. They used this strategy to propose wait-free implementations of inked-list, skip-list and binary search trees. There have been prior work on wait-free queues and stacks [11], [2]. However, to our knowledge, this is the first work on a wait-free implementation of an abstract data type that supports add, remove, search and range queries.

8 Conclusion

We developed an efficient concurrent data structure Uruv that supports wait-free addition, deletion, membership search and range search operations. Theoretically, Uruv offers a finite upper bound on the step complexity of each operation, the first in this setting. On the practical side, Uruv significantly outperforms the existing lock-free B$^+$Tree variants and a recently proposed linearizable lock-free range search algorithm.

References

1. Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. *ACM SIGPLAN Notices*, 53(1):14–27, 2018.
2. Hagit Attiya, Armando Castañeda, and Danny Hendler. Nontrivial and universal helping for wait-free queues and stacks. *Journal of Parallel and Distributed Computing*, 121:1–14, 2018.
3. Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *PPOPP*, pages 357–369, 2017.
4. Gaurav Bhardwaj, Abhay Jain, Bapi Chatterjee, and Sathya Peri. Wait-free updates and range search using uruv, 2023. `arXiv:2307.14744`.
5. Anastasia Braginsky and Erez Petrank. A lock-free b+ tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 58–67, 2012.
6. Trevor Brown. Techniques for constructing efficient lock-free data structures. *CoRR*, abs/1712.05406, 2017. URL: `http://arxiv.org/abs/1712.05406`, `arXiv:1712.05406`.

7. Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *International Conference On Principles Of Distributed Systems*, pages 31–45. Springer, 2012.
8. Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *PPOPP*, pages 276–291, 2020.
9. Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, pages 1–10, 2017.
10. Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.
11. Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 325–334, 2011.
12. flurry. Flurry Analytics. `https://www.flurry.com/`, 2022. Online; accessed May 2022.
13. Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing: 15th International Conference, DISC 2001 Lisbon, Portugal, October 3–5, 2001 Proceedings 15*, pages 300–314. Springer, 2001.
14. Maurice Herlihy and Nir Shavit. On the Nature of Progress. In *OPODIS*, pages 313–328, 2011.
15. Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
16. Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *ACM SIGPLAN Notices*, 47(8):141–150, 2012.
17. Thomas Kowalski, Fotios Kounelis, and Holger Pirk. High-performance tree indices: Locality matters more than one would think. In *11th International Workshop on Accelerating Analytics and Data Management Systems*, 2020.
18. Aravind Natarajan, Lee Savoie, and Neeraj Mittal. Concurrent wait-free red black trees. In *Safety-critical Systems Symposium*, 2013.
19. Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. Bundled references: an abstraction for highly-concurrent linearizable range queries. In *PPOPP*, pages 448–450, 2021.
20. Erez Petrank and Shahar Timnat. A practical wait-free simulation for lock-free data structures. 2017.
21. Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. Latency critical big data computing in finance. *The Journal of Finance and Data Science*, 1(1):33–41, 2015.
22. Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-Free Linked-Lists. In *OPODIS*, pages 330–344, 2012.
23. Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488, 2018.
24. Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *PPOPP*, pages 31–46, 2021.
25. Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.