Skip Hash: A Fast Ordered Map Via Software Transactional Memory

Anonymous Author(s)

***Abstract*—Scalable ordered maps must ensure that range queries, which operate over many consecutive keys, provide intuitive semantics (e.g., linearizability) without degrading the performance of concurrent insertions and removals. These goals are difficult to achieve simultaneously when concurrent data structures are built using only locks and compare-and-swap objects. However, recent innovations in software transactional memory (STM) allow programmers to assume that multi-word atomic operations can be fast and simple.**

This paper introduces the skip hash, which uses STM to combine a skip list and a hash map behind a single ordered map abstraction, resulting in $O(1)$ overhead for most operations. The skip hash makes use of a novel range query manager—again leveraging STM—to achieve fast, linearizable range queries that do not inhibit scalability. In performance evaluation, we show that the skip hash outperforms the state of the art in almost all cases. This places the skip hash in the uncommon position of being both exceedingly fast and exceedingly simple, which demonstrates that designing novel STM-based data structures is a promising direction for future research.

***Index Terms*—Synchronization, Concurrent Data Structures, Range Queries, Software Transactional Memory**

I. Introduction

Ordered maps are one of the most fundamental data structures for modern data-intensive applications. These data structures implement *elemental* operations for creating, updating, clearing, and querying associations between keys and values. They also support *point query* operations such as `floor`, `ceil`, `pred`, and `succ`, which find the closest key in the map that is $\leq$, $\geq$, $<$, or $>$ a given key, respectively. Finally, they support *range query* operations, which gather all key/value pairs whose keys fall within a range specified by $[l, r]$.

Applications must employ many threads to process large amounts of data quickly, and so they must use ordered maps that scale well. However, it is challenging to implement efficient range queries that provide a reasonable correctness property (e.g., linearizability [1]). Consider an execution history in which one thread executes `range(l, r)` while other threads repeatedly try to insert and remove keys between `l` and `r`. If the range query does not block these elemental operations, then how can it ensure that its result corresponds to the map's state at some single point in time?

Prior work has largely coalesced around two ideas. The first employs optimistic synchronization techniques [2] so that elemental operations do not induce artificial contention when reading data structure nodes. The second attaches some manner of versioning to map entries, so that range queries can operate over a "frozen" version of the data structure while concurrent elemental operations update a "fresh" copy.

The key insight in this paper is that modern software transactional memory (STM) systems have become surprisingly efficient [3]–[7]. This is important, because STM simplifies two common synchronization behaviors: (B1) modifying several locations as a single, indivisible operation and (B2) performing an action only if some variable remains unchanged throughout that action's duration.

In this paper we design a map from scratch using STM as its primary synchronization mechanism, to show that STM enables new approaches to concurrent data structure design that are simple and perform extremely well. Our first observation is that B1 lets us compose a hash map with a *doubly linked* skip list. We call this data structure a "skip hash." The skip hash scales well, has $O(1)$ complexity for most elemental operations, and is easy to verify. However, it is prone to starvation for long-running range queries. We remedy this by introducing a new mechanism for coordinating range queries and elemental operations. This mechanism adds $O(1)$ overhead to elemental operations (typically only a single read, leveraging B2). However, it has high contention when range queries are small or many removals execute concurrently with a range query. Therefore, we employ a fast-path/slow-path strategy [8], so that the range query coordinator is only used as a fallback, alleviating this contention.

On standard microbenchmarks, the skip hash outperforms the current state of the art on almost all configurations, often by a large margin. In addition, since the skip hash uses STM, its implementation is free from complex and error-prone synchronization. Furthermore, STM makes it trivial for the skip hash to support complex key and value types larger than one memory word in size.

This paper makes the following contributions:

- We show that composing data structures with STM can provide high performance while retaining key benefits of STM, such as easy reasoning about correctness and support for complex data types.
- We further show that the transactional composition allows concurrent data structures to easily achieve *asymptotic* improvements that would be difficult to achieve with traditional techniques.
- Finally, we introduce a technique for efficient range queries, based on a novel use of STM.

These results suggest that STM should have a prominent place in future concurrent data structure research and practice.

The remainder of this paper is organized as follows. In Section II, we discuss related work in range queries and STM. In Section III, we present a simple design for a composite

data structure that supports elemental operations. Next, in Section IV, we extend this design to support linearizable range operations. Then, in Section V, we evaluate an implementation of this design, and show that it performs competitively with the state of the art. Finally, Sections VI and VII discuss future work and conclude.

II. Related Work

A. Range Queries

One of the main advantages of ordered maps is their ability to efficiently support range queries. There is a significant body of work on how to implement these queries in a concurrent setting [9]–[15], including a recent focus on aggregate range queries [16]–[18]. Early works in this area often relied on restarting a range query when a concurrent update occurred. One particularly relevant study [19] used STM for this purpose. However, at the time, STM implementations were neither highly efficient nor expressive, which limited scalability, especially in experiments with a high update rate.

Arbel and Brown utilized an epoch-based reclamation (EBR) scheme in their range queries [10]. The key idea is to assign a timestamp to each range query, enabling it to determine whether an element existed before it began. The main challenge arises with removal operations, as a needed element may be missed by the traversal if it is concurrently removed. Their technique leveraged EBR, in which removed elements are temporarily stored in a "limbo set," to detect overlooked elements after the range query completed. This concept is, in a sense, the inverse of our design, in which removal of elements is *delegated to* range queries, thereby avoiding a search through the limbo set.

The approach in most other prior work, such as vCAS [11] and Bundling [13], is based on Multi-Version Concurrency Control (MVCC). In these methods, updates to a data structure do not overwrite the previous versions of updated elements, links, etc. Since space overheads can grow significantly, custom garbage-collection-like mechanisms are then used to reclaim versions when they are no longer useful. One work that follows this approach [12] is particularly interesting because it uses hash-tries as the foundation instead of the usual ordered sets. However, the issue with hash tries and similar structures [12], [20]–[23] is that they can only support range queries if the key itself is used as the hash. This is not always feasible and can result in an unbalanced structure. Among these works, [23] employs an idea similar to ours, using a hash map as a cache, but in a different context.

B. Modern STM Systems

TM was originally proposed as a technique for simplifying the creation of concurrent data structures [24], [25]. While there was considerable effort to expand TM's scope so that it could serve as the foundation of a full-fledged programming model [26], there have always been arguments that it should be thought of as merely an "implementation technique" [27]. This has two main benefits. First, when programmers can constrain the scope of the TM, they can guarantee that transactional data is never accessed outside of transactions. The resulting transactional semantics [28] avoid "privatization" overheads [29]. Second, if programmers are allowed to know about the TM's internal implementation, they can employ many low-level optimizations [30] to reduce latency.

In recent years, several STM systems have explored this design space, including TinySTM [7], MCMS [4], Lock-Free Locks [31], PathCAS [3], exoTM [6], and the slow clock STM [32]. While these systems differ with regard to the progress guarantees they offer, they coalesce around the following design principles:

- STM algorithms based on ownership records (orecs) [33], [34] scale best for map data structures.
- The use of a global clock within the STM algorithm [33] need not be a significant bottleneck [32], [35], [36].
- Acquiring orecs upon first write access [34], [37], [38] and leveraging undo logging results in the lowest latency.
- Static read-only transactions can be optimized to negligible overhead [33].
- Orecs should be co-located with the objects they protect, not kept in a separate table [3], [4], [6], [39].
- Programmers should optimize instrumentation for `const` fields and use specialized API calls to avoid redundant logging [40]–[42].
- STM instrumentation should be in-lined into the program [3], [32], [43].

Finally, these works have collectively shown that STM can be fast, even without hardware acceleration. This is important, since HTM is prone to security vulnerabilities and performance pathologies. Fortunately, in this paper we find that STM can enable very fast concurrent data structures.

III. A Fast Ordered Map

The best-performing lock-free map implementations use compare-and-swap objects [44]–[47]. To simplify linearization arguments, these maps are singly linked. That is, nodes in tree-based maps tend not to have parent pointers, and nodes in skip lists tend not to have predecessor pointers. However, STM makes it straightforward to implement double-linking and to access multiple data structures in a single, indivisible operation. Given these benefits of STM, Figure 1 defines the skip hash as the composition of a doubly linked skip list that maps keys to values, and a closed-addressing hash table that maps keys to skip list nodes.

The skip list is implemented as a sequence of n objects of type `sl_node` (line 1). Each contains a key/value pair (k, v), a height h, and a "tower" consisting of h pairs of pointers. Upon each node's insertion, h is generated using a geometric distribution with $p = 1/2$ in the range $[1, m]$ where $m \geq \lg n$. The skip list has m levels, where each level l represents a doubly linked list whose traversal "skips" all nodes where $h \leq l$. Sentinel head and tail nodes (lines 9–10) of height m, with keys $\perp$ and $\top$ respectively, bookend the skip list.

The skip hash is designed with the invariant that the set of keys present in the skip list and hash map is identical at all times. The use of this property asymptotically accelerates

```
1  type sl_node<K, V>      # A node in the skip list
2    const key: K          # The (immutable) key
3    val: V                # The associated value
4    const height: u8      # Height of "neighbors" (min 1)
5    # Array of predecessor/successor links at each level
6    neighbors: Array<(sl_node<K, V>, sl_node<K, V>)>
7
8  type skiplist<K, V>
9    head: sl_node<K, V> # Sentinel nodes initially stitched
10   tail: sl_node<K, V> #   together at all levels
11
12 type skip_hash<K, V>
13   map: hashmap<K, sl_node<K, V>>
14   sl: skiplist<K, V>
15
16 func lookup(k: K) -> Option<V>:
17   atomic:
18     # If the key is present, the map routes to the node
19     let n = map.get(k)
20     if (!n) return None else return n.val
21
22 func remove(k: K) -> bool:
23   atomic:
24     # If the key is not present, stop at map lookup
25     let n = map.get(k)
26     if (!n) return false
27     # Otherwise remove from the map, and leverage
28     #   double-linking to avoid skip list traversal
29     map.remove(k)
30     n.unstitch()
31     return true
32
33 func insert(k: K, v: V) -> bool:
34   atomic:
35     # O(1) if the key is already present
36     if (map.get(k)) return false
37     # O(log n) with optimized skip list insert,
38     #   because k is necessarily absent
39     let new_node = sl.insert_optimized(k, v)
40     # Finally, update the map to reference the new node
41     map.insert(k, new_node)
42     return true
43
44 func ceil(k: K) -> K: # Find smallest key >= k
45   atomic:
46     if (map.get(k)) return k # O(1) if key present
47     return sl.succ(k).key    # O(log n) otherwise
48
49 func succ(k: K) -> K: # Find smallest key > k
50   atomic:
51     let node = map.get(k)
52     if (node) return node.neighbors[0].second.key
53     return sl.succ(k).key
```

Fig. 1: Transactional Composition of an Unordered Map and Skip List

several skip list operations. Furthermore, we assume without loss of generality that all hash map operations are $O(1)$. Therefore, we observe that `lookup(k)` (line 16) is always $O(1)$, consisting of a map lookup and then at most one additional read. When k is absent, `remove(k)` is also $O(1)$, returning immediately after the failed map lookup (line 26). The same logic explains why `insert(k)` (line 33) and point queries are all $O(1)$ when k is present. This can be seen in `ceil(k)` (line 44) and `succ(k)` (line 49); `floor(k)` and `pred(k)` can be implemented similarly.

When k is present, `remove(k)` avoids an $O(\log n)$ skip list traversal by using the map to find the node with key k (line 25). Since the skip list is doubly linked, the predecessor and successor at each level can be located in $O(1)$ time, so unstitching any one level has constant overhead. The expected height of a randomly selected node is $1/p = 2$, so the average complexity is $O(1)$, with a worst case of $O(\log n)$.

Thus, the only operations that cannot avoid an $O(\log n)$ skip list traversal are `insert(k)` and the four point queries, and only when when k is absent. Point queries then perform $O(1)$ additional reads, and `insert()` performs up to $O(\log n)$ writes. However, the expected height is $O(1)$, like with `remove()`; and since the key is guaranteed to be absent, the skip list insertion logic can be optimized (line 39). Note that the presence of predecessor pointers doubles the number of writes compared to a singly linked skip list.

To summarize, STM lets us implement the map interface by composing two data structures. The resulting data structure avoids skip list traversal in most cases, and thus the majority of its operations only perform a constant number of reads and (expected) writes.

Since the skip hash uses a modern STM as described in Section II, its ability to scale hinges on the probability of concurrent operations accessing the same datum simultaneously, with at least one performing a write. Because transactions in a skip hash must validate after performing a write, they have a larger contention window than a skip list implemented with lock-free or optimistic locking techniques. There is a surprising caveat, however: `remove()` operations do not read any skip list node that they do not also write. Therefore, `remove()` is not vulnerable to the sorts of artificial aborts that TM is known to induce. The only situation where unnecessary conflicts can arise, then, is between pairs of insertions. Our evaluation shows that this is not a significant concern.

IV. Supporting Range Queries

Given our use of STM, the simplest implementation of a linearizable range query is to execute the entire query as a single transaction. This approach avoids introducing additional global synchronization metadata, so range queries will only conflict with concurrent overlapping insertions and removals. Such conflicts should be uncommon in low-skew workloads where the data structure is highly populated, updates are rare, or range queries are short. In workloads where conflicts are more common, however, STM-based range queries may require many attempts to succeed. This can significantly degrade performance or even lead to the sort of starvation that cannot be solved through traditional contention management [48].

Therefore, we introduce a new object in this section: the range query coordinator (RQC). The RQC assigns version numbers to range queries and skip list nodes. These versions allow a range query to ignore nodes inserted after or removed before it began. We can then execute a range query as a sequence of transactions, each of which accesses a small number of consecutive skip list nodes.

While this approach does not alter the implementation of `lookup()` and only trivially changes `insert()`, it creates a new burden for `remove()`: While a node can be "logically removed" through the use of version numbers, it cannot be unstitched if a concurrent range query needs it. Therefore, we introduce the concept of "safe nodes", which cannot be

unstitched during an in-flight range query and which can serve as boundary points for a range query's transactions. We also introduce a deferral mechanism through which `remove()` delegates unstitching to an in-flight range query. While delegation only adds $O(1)$ overhead, it can increase contention. We reduce contention via a fast-path strategy: Every range query first tries to complete along the fast path—finishing the entire query in a single transaction—before falling back to the slow path. On the slow path, it uses one transaction to acquire a unique version number, a sequence of transactions to access the pairs in the map, and then a finalizing transaction. By keeping transactions short, the range query is thus able to avoid contention and make progress.

A. An Abstract Coordinator

The foundation of our slow path is the RQC, which has two obligations. First, it produces monotonically increasing version numbers which order slow-path range queries with respect to successful insertions and removals. Second, it determines when it is safe to unstitch and reclaim logically deleted nodes. It does this through an interface of consisting of four methods:

- `on_range()` – Registers a new slow-path range query and assigns a unique version number to it.
- `on_update()` – Reports the most recent range's version number to the calling `insert()` or `remove()`.
- `after_remove(sl_node)` – Unstitches and reclaims a logically deleted skip list node immediately if safe; otherwise, schedules it for deferred removal.
- `after_range(ver)` – Marks the range query with version number `ver` complete. May also unstitch and reclaim nodes whose removal was deferred by concurrent calls to `after_remove()`.

B. Logical Deletion

We augment the `sl_node` type with two new fields. `n.i_time` is an immutable value that records the version number of last range query that began before n's insertion. `n.r_time` is initially `None`, indicating n is logically present. To *logically delete* n, this field is set to the most recent range query's version. n may remain "physically" linked into the skip list for some time afterwards to allow it to be processed by that range query and/or its predecessors. For any other purpose, n is considered to be absent from the data structure.

While nodes may remain in the skip list for some time after their logical deletion, they are removed from the hash map immediately. This provides a powerful invariant: The hash map always reflects the current logical state of the data structure. This means no changes are needed to the `lookup()` operation. When point queries do not find their operand in the map and therefore must perform a lookup in the skip list, they require a single-line edit to check `r_time` and ensure they do not return a logically deleted node. The `insert()` and `remove()` operations require slightly more invasive changes, which are shown in Figure 2.

Leveraging the invariant described above, `remove(k)` begins by querying the map for a node with key k (line 3).

```
1  func remove(k: K) -> bool:
2    atomic:
3      let n = map.get(k)
4      if (!n) return false # Failed; key absent
5      map.remove(k)
6      n.r_time = rqc.on_update() # Logically delete
7    rqc.after_remove(n)
8    return true
9
10 func insert(k: K, v: V) -> bool:
11   atomic:
12     let n = map.get(k)
13     if (n) return false # Failed; key already present
14     # The key may be present in the skip list,
15     #   but only in a logically deleted node
16     let new_node = sl.insert_after_logical_deletes(k, v)
17     new_node.i_time = rqc.on_update()
18     map.insert(k, rqc)
19     return true
```

Fig. 2: Elemental operations of a skip hash augmented with a range query coordinator.

If k is absent from the map, the operation completes and returns `false` (line 4). Otherwise, the operation removes the node from the map to maintain the invariant (line 5) and logically deletes the node by setting `r_time` to the current version number, as reported by `on_update()` (line 6). At this point, subsequent elemental operations will not find the key, and concurrent range queries will use the node's version number to decide whether to process it. All that remains is to unstitch and reclaim the node. This is accomplished via `after_remove(n)`, which may choose to delegate clean-up to an ongoing range query (line 7).

Relative to Figure 1, `insert(k)` changes in two ways. When the hash map indicates that k is absent, the operation uses `on_update()` to initialize the new node's `i_time` (line 17). Additionally, the logic used to insert that node into the skip list must be modified slightly, because one or more logically deleted nodes with key k may still be present in the skip list. So, instead of completing as a failed insertion upon finding the key present in the skip list, it will instead insert the new node *after* any logically deleted nodes with key k.

C. Safe Nodes

In Figure 2, `insert()` and `remove()` can modify the skip list at all times. While TM handles any conflicts that occur during a transaction, slow-path range queries must tolerate skip list changes when they are *between* transactions. Suppose an ongoing slow-path range query R commits a transaction partway through its range and stops on some node n. If a concurrent `remove()` operation deletes n, this could result in R accessing freed memory or other erroneous behavior. To prevent this, we ensure that slow-path range queries only stop on *safe nodes*—nodes guaranteed not to be unstitched or reclaimed during their execution.

For a query R on range `(l,r)` with version number `ver` to be correct, it must process every node in its *processing set*, defined as every node `n` with the following properties:

1) $\texttt{l} \leq \texttt{n.key} \leq \texttt{r}$ (i.e., `n`'s key is in R's range)
2) $\texttt{n.i\_time} < \texttt{ver}$ (i.e., `n` was inserted before R began)

3) `n.r_time` = `None` ∨ `n.r_time` ≥ `ver` (i.e., `n` was not deleted before R began)

For R to linearize, `n` must remain in the skip list until R has progressed past it. We adopt a stricter condition that requires less inter-thread communication: `n` cannot be removed until R completes. Thus, every node in R's processing set is also a safe node where R can stop between transactions. Additionally, any node n which violates condition (1) but satisfies conditions (2)[1] and (3) is also considered safe, despite being outside of the processing set. Lastly, as the head and tail sentinels are never removed from the skip list, they are considered safe.

For any removed node n, if n is needed by an ongoing slow-path range query R, the RQC defers n's removal until R's invocation of `after_range()` at the earliest. In doing so, the RQC ensures that n is safe for R. This is why the RQC is responsible for unstitching and reclaiming nodes.

As soon as a range query R commits a transaction in which it called `on_range()`, the set of safe nodes for R is fixed—no nodes can enter or leave it. Any newly inserted node will have `i_time` > `ver`, violating condition (2). Any removed safe node will have its `r_time` change from `None` to a value ≥ `ver`, preserving condition (3). Thus a range query does not need to access every node in its processing set in a single transaction. Instead, it can split its operation into several transactions, each beginning and ending on a safe node.

D. Implementing the Range() Operation

The full `range()` algorithm is presented in Figure 3. `range(l,r)` takes two keys representing the bounds of the range (line 2). First, the algorithm tries the fast path a few times (line 4). In our implementation, we set `FAST_PATH_TRIES` to 3. If that threshold is exceeded, it falls back to the slow path (line 7).

1) Fast Path: `range_fast()` (line 15) attempts to execute the range query as a single transaction. It does not need to worry about safe nodes and is not assigned a version number, so its logic is relatively simple. We use `atomic(try_once)` to indicate that the transaction should not retry if it aborts (line 16). This lets the caller fall back to the slow path after `FAST_PATH_TRIES` attempts fail.

The fast path first uses the point query `sl.ceil(l)` to find the first logically present node at or after the start of the range (line 18).[2] It then scans the skip list, copying the keys and values of all logically present nodes (line 20) until it reaches a node beyond the end of the range (possibly `sl.tail`).

2) Slow Path: The slow path is more complex. First, a transaction is used to initialize the range query (line 7). In addition to finding the start node using `sl.ceil(l)`, it also calls `rqc.on_range()`. This informs the RQC of its existence and gets a version number, `ver`. Doing both tasks in one transaction ensures that `start` will be a safe node.

[1]The RQC may immediately unstitch nodes inserted after the most recent range query, so a node that violates condition (2) is not necessarily safe.

[2]We say "at or after the start of the range" rather than "within the range" because it is possible that there are no nodes within the range.

```
1  # Process all nodes with a key in the range [l, r]
2  func range(l: K, r: K) -> Set<(K,V)>:
3    # Try the fast path a few times
4    for i in 0 .. FAST_PATH_TRIES:
5      let set = range_fast(l, r)
6      if (set != None) return set
7    atomic:  # Fall back to slow path
8      let start = ceil(l)
9      let ver = rqc.on_range()
10   let set = range_slow(start, r, ver)
11   rqc.after_range(ver)
12   return set
13
14 # Try to complete a fast-path range op
15 func range_fast(l: K, r: K) -> Option<Set<(K,V)>>:
16   atomic(try_once): # does not retry on conflict
17     let set = {}
18     let n = sl.ceil(l)
19     while n.key <= r:
20       if (n.r_time == None) set.add((n.key, n.val))
21       let n = n.neighbors[0].second
22     return set
23   return None
24
25 # Process nodes in range for a slow-path range query
26 func range_slow(n: sl_node, r: K, v: u64) -> Set<(K,V)>:
27   let set = {}
28   # this atomic block won't undo changes to n/set
29   atomic(no_local_undo):
30     while n.key <= r:
31       let next = next_safe(n, v)
32       set.add((n.key, n.val))
33       n = next
34   return set
35
36 # Find the next safe node after n for RQC value ver
37 func next_safe(n: sl_node, ver: u64) -> sl_node<K,V>:
38   let n = n.neighbors[0].second
39   while !is_safe(n, ver) n = n.neighbors[0].second
40   return n
41
42 # Determine if the given node is safe
43 func is_safe(n: sl_node, ver: u64) -> bool:
44   if (n == sl.head || n == sl.tail) return true
45   if (n.i_time >= ver) return false
46   return (n.r_time == None || n.r_time >= ver)
```

Fig. 3: Range query implementation

After this setup is complete, `range_slow()` (line 26) is responsible for collecting key/value pairs. The variables `set` and `n` record the progress of the range query—the set of collected pairs and the current node, respectively. We use the syntax `atomic(no_local_undo)` to indicate that the STM should not roll back modifications to these local variables upon transaction abort (line 29). Due to the fact that `range_slow()` does not roll back these local variables or modify any shared variables, it is able to keep all progress up to the point where the transaction aborted—thus turning aborts into unplanned early commits. Furthermore, `n` is only ever set to a safe node, so transactions are always able to start where the last one ended. The somewhat verbose syntax on lines 30–33 employs the clearly defined abort points of STM to ensure no key/value pair is inserted into the set twice.

`next_safe()` (line 37) traverses the bottom level of the skip list until it finds the next safe node after `n`. Since `sl.tail` is always safe, such a node always exists. `is_safe()` (line 43) determines if a node is safe for a given range query, using the criteria from Section IV-C.

On line 11 the slow-path range query informs the RQC

```
1  type rqc # The RQC implementation
2    counter: u64 # Counter for generating version numbers
3    range_ops: LinkedList<range_op>
4
5  type range_op # Metadata about a slow-path range op
6    const ver: u64 # Version number of the range op
7    deferred: LinkedList<sl_node> # Nodes to remove
8
9  # Inform the RQC about a slow-path range query
10 func on_range() -> u64:
11   range_ops.append(new range_op(++counter, []))
12   return counter
13
14 # Get a version number for insertion/removal time
15 func on_update() -> u64:
16   return counter
17
18 # Give a node to the RQC for removal when safe
19 func after_remove(n: sl_node):
20   atomic:
21     let tail = range_ops.tail() # Returns null if empty
22     if !tail || n.i_time >= tail.ver:
23       n.unstitch() # Safe to remove immediately
24       delete n
25     else:
26       tail.deferred.append(n) # Defer removal
27
28 # Clean up after a slow-path range op
29 func after_range(ver: u64):
30   let removals = [] # Nodes to remove immediately
31   atomic:
32     let op = range_ops.find(ver) # Find by version num.
33     let pred = range_ops.pred(op) # Point query
34     range_ops.remove(op)
35     if (!pred): # Take responsibility for removing now
36       removals = op.deferred
37     else: # Defer removals further
38       pred.deferred.append_all(op.deferred)
39     delete op
40   for n in removals: # Remove the deferred nodes
41     atomic:
42       n.unstitch()
```

Fig. 4: A concrete implementation of the range query coordinator.

that it is complete by calling `rqc.after_range()`. This tells the RQC that it no longer needs its safe nodes, which may trigger the unstitching and reclamation of nodes whose removal was previously deferred.

E. Implementing the Range Query Coordinator

Figure 4 presents a concrete implementation of the RQC. The RQC itself (line 1) has two fields: `counter`, a 64-bit integer for generating version numbers; and `range_ops`, a doubly linked list of all ongoing slow-path range queries. The list contains objects of type `range_op` (line 5), which consist of two fields: a version number (`ver`) and a set of logically deleted nodes whose removal has been deferred until after the range query is complete (`deferred`).

The simplest algorithm would assign a unique version number to each operation, but `counter` risks becoming a contention hotspot, with every write to it potentially causing aborts for concurrent transactions that interacted with it. Instead, we only increment the counter for range queries (`on_range()`, line 11). Elemental operations reuse the most recent range query's version number (`on_update()`, line 16), thus ordering themselves after it. `on_range()` also appends a new `range_op` object to the list.

While consecutive range queries could also share version numbers, we did not see any benefit to doing so: Short range queries should complete on the fast path without accessing the RQC, and long range queries should perform enough real work that their RQC increments should not contend.

In `after_remove()` (line 19), the RQC assumes responsibility for unstitching and deleting a logically removed node n. First, it checks if the removal can be done immediately (line 22). It can do so if either `range_ops` is empty or n was inserted after the most recent ongoing slow-path range query R_{last}. Otherwise, n is a safe node for R_{last}, so n is added to its `deferred` list (line 26).

The most complex logic is in `after_range()` (line 29). First, the `range_op` representing the finishing range query is removed from the list (line 34). Then, deferred removals must be handled. There are two scenarios here. (1) If the removed `range_op` is the oldest one (the head of `range_ops`), then the nodes in `deferred` can now be safely unstitched and reclaimed (lines 36, 40–42). (2) However, if a `range_op` with an earlier version number still exists, one of its safe nodes may be in `deferred`. The potential benefit of scanning the entries in `deferred` and comparing their removal times to the remaining `range_ops`' versions is not worth the overhead. Instead, the removal of all of these nodes is deferred further by adding them to the predecessor's `deferred` list, via an $O(1)$ list append operation. (line 38) Since these nodes are being passed backward, not forward, every node is guaranteed to be reclaimed eventually.

When many removals run concurrently with a slow-path range query R, insertions into R's `deferred` list can become a contention bottleneck. By virtue of the above mechanism, it is safe to delegate unstitching and reclaiming a range query's safe node to a later range query. We leverage this fact by keeping a buffer of removed nodes for each thread. We can then modify line 26 in `after_range()` to instead push the node into that thread's buffer. When the buffer is full (size 32 in our implementation), the thread checks if there are any active slow-path range queries. If not, it can immediately unstitch and reclaim all entries. Otherwise, it transfers the entire buffer to the most recent range query's `deferred` list via an $O(1)$ append operation.

F. Correctness

Here we briefly outline the properties that would underpin a correctness proof. First, we observe that each elemental and point query operation executes as a single transaction. Thus as long as each would be correct in a sequential implementation, they are all correct in a concurrent execution. Next, we note that a fast-path range query works correctly because it is also a single transaction. The node found by `ceil` will not be unstitched, nor can nodes within the range be added or removed, without causing the entire range query to abort.

The correctness of a slow-path range query R is more complex. The first atomic step uses `ceil` to find a logically present node and then increments the version number. This ensures that the node will not be unstitched before R completes.

This is also R's linearization point. Then, R only pauses at nodes that cannot be unstitched until after it completes. The unstitching of an unsafe node while R is accessing it would cause a transaction abort, causing R to retry from the last safe node. Since all the unstitched nodes have a removal time less than the version number of our range query, the query will not miss any necessary nodes. Finally, the `i_time` field ensures that the range query does not include nodes inserted after its linearization point.

V. Evaluation

In this section, we compare the performance of the skip hash against a set of ordered map implementations that represent the current state of the art. These include a binary search tree and skip list based on the versioned compare-and-swap (vCAS) technique [11], and a skip list that uses bundled references [13]. We also considered variants of these three maps that utilize a hardware timestamp counter instead of a shared memory counter, using the x86 `rdtscp` instruction [49]. This eliminates a known contention bottleneck for vCAS and bundling. This optimization improves performance in all evaluated workloads, and so we have excluded the non-optimized variants of these maps from all charts. Lastly, in workloads consisting entirely of elemental operations, we evaluate a hashmap and doubly linked skip list that do not support range queries, implemented using STM.[3]

A. Experimental Setup

All experiments were conducted on a system running Ubuntu 22.04 with 180 GB of RAM and two Intel Xeon Platinum 8160 CPUs running at 2.10 GHz. Each CPU has 24 cores, for a total of 48 cores and 96 hardware threads. For all maps evaluated, threads were pinned to cores in the same way. The first 24 threads were pinned to unique cores on a single CPU; the next 24 threads were pinned to the second thread on those cores; and the last 48 threads were distributed in the same manner on the second CPU. Thus, in all charts, symmetric multithreading is not a factor for data points up through 24 threads, and non-uniform memory access (NUMA) latency is not a factor up through 48 threads.

We used the evaluation framework from [49], which already included support for the vCAS and bundling-based data structures, as well as their `rdtscp`-enhanced variants. The default behavior of this framework is that worker threads perform lookups, insertions, removals, and range queries in proportions that vary by workload. Keys and values were both represented as signed 64-bit integers. In this paper, we report results for a key universe of 10^6. In experiments with larger universes, the trends are the same, but the separation between the skip hash and other data structures is more pronounced, particularly for elemental operations.

Before each experiment, each map was pre-filled with half of the keys in the universe, resulting in a population of $5 \cdot 10^5$. Elemental operations chose keys uniformly at random from the universe. Range queries did the same to choose l and computed r by adding a fixed range length determined by the particular workload. They then copied all keys and values within that range into a pre-allocated buffer. In all workloads, update operations were evenly split between insertions and removals, so that the population of the data structure remained roughly constant at all times with high probability. This also meant that all elemental operations were equally likely to succeed as fail, and each range query was expected to process $(r - l)/2$ entries. Unless stated otherwise, the range length was 100 keys (as in [13], [49]), for an expected 50 entries processed. All benchmark and map code was written in C++ and compiled with g++ version 12.3.0 using flags `-O3 -std=c++20`. All experiments used the jemalloc [51] allocator. All data points are the average of five 3-second trials. Error bars are omitted, because we did not observe significant variance.

In the experiments, we considered three implementations of the skip hash: one where all range queries use the fast path, one where they all use the slow path, and one where they fall back to the slow path after three fast-path failures. Following common guidance that hash tables perform best when about 70% full and that prime hash table sizes are advantageous [52], we configured each skip hash's hash table to consist of $714,341$ buckets. This number was chosen due to being the smallest prime that yields a hash table utilization rate $\leq 70\%$ for the expected population. Keys were hashed using `std::hash`. For the skip hash and skip lists, the tower height was set to 20, as 2^{20} is slightly greater than 10^6.

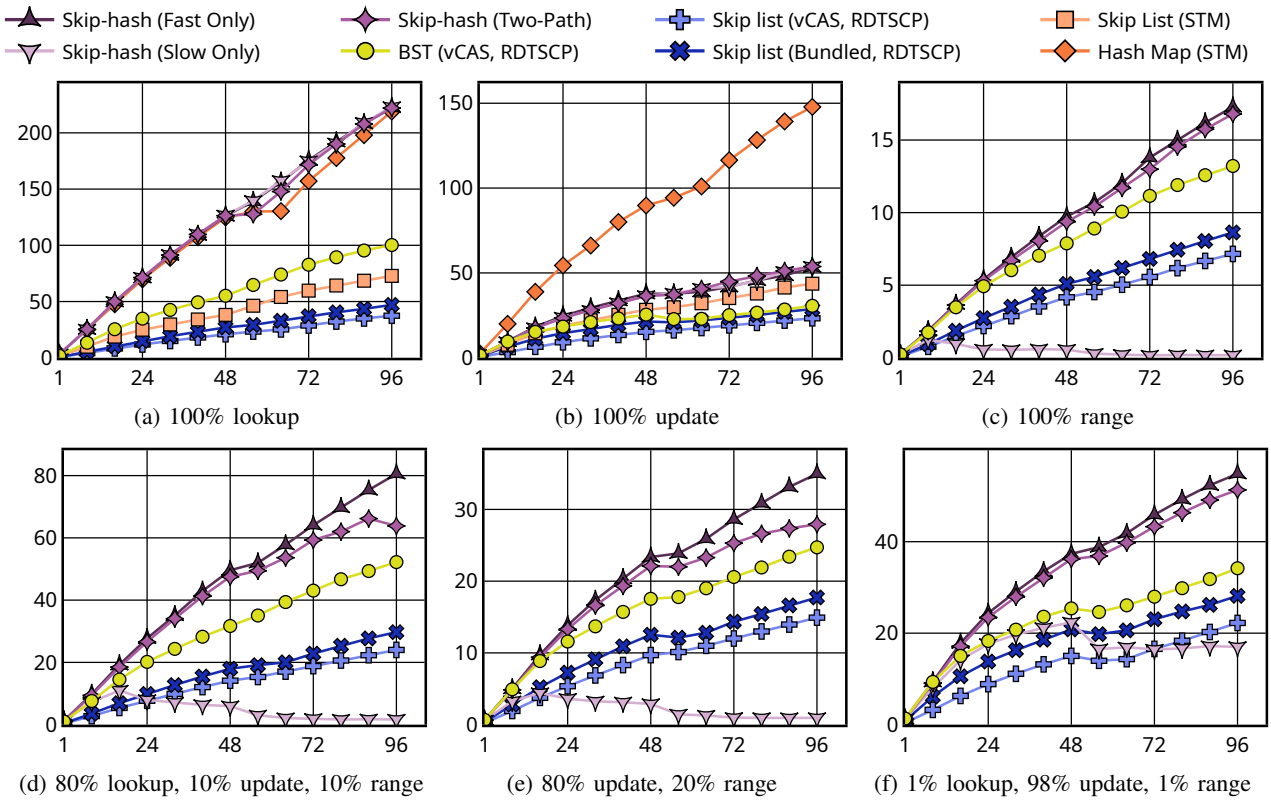
The STM implementation used in the skip hash, skip list, and hash map is exoTM [6]. As we expect conflicts to be rare, we chose the eager (undo) algorithm *without* timestamp extension. This is expected to have the lowest latency, but also the highest writer abort overhead of any STM algorithm in the exoTM framework. Our charts present results for an `rdtscp`-based clock [36]. We also tested the gv1 and gv5 logical clocks [33], [35]. Gv1 scaled poorly for the skip hash's small transactions, while gv5 and the slow clock suffered in slow-path experiments, where our RQC implementation violates the assumptions that underpin their designs.

B. Analysis

Figure 5 presents throughput as the thread count is varied from 1 up to the maximum hardware thread count in a microbenchmark with various workloads. We begin by discussing workloads that perform a single operation in isolation to evaluate the raw performance of those operations.

1) Isolated Workloads: In a workload consisting of 100% lookup operations (Figure 5a), the skip hash scales to the maximum thread count and achieves a $> 2\times$ speedup over all data structures except the hash map, which does not support range queries. This is because the `lookup()` operation is $O(1)$ for the skip hash and hash map, as opposed to $O(\log n)$ for skip lists and BSTs. Furthermore, our STM implementation does not experience aborts in this read-only workload. We also observe that the STM skip list outperforms skip lists with more complex synchronization mechanisms, which confirms

[3]More experiments are available in the supplementary material [50].



(a) 100% lookup (b) 100% update (c) 100% range

(d) 80% lookup, 10% update, 10% range (e) 80% update, 20% range (f) 1% lookup, 98% update, 1% range

Fig. 5: Comparison of skip hash performance versus state-of-the-art ordered maps, with varying mixtures of lookup, update, and range operations. The x-axis represents thread count, and the y-axis represents throughput as millions of operations per second. All range queries were of length 100, thus processing 50 keys on average.

the claim that modern STM can be fast, and shows the benefit of composing a hash map and skip list.

Next, in Figure 5b, we evaluate the performance of update operations (an equal mix of insertions and removals). The throughput of the skip hash drops due to the additional overhead of stitching and unstitching (and rare transaction aborts). However, it maintains a significant lead over all data structures save the hash map, as hash acceleration allows the skip hash to avoid $O(\log n)$ skip list searches for all operations except successful insertions. Furthermore, we see that the slow path mechanism does not introduce significant overhead when it is not in use. Note that in these charts and many that follow, a small performance dip occurs for some maps after 48 threads, due to the cross-chip memory traffic.

Lastly, we evaluate the performance of range queries of length 100 in Figure 5c. Note that in this figure, the y-axis represents completed range queries, not the total number of nodes accessed. As this is a read-only workload, no transactions will abort, so the two-path variant will complete all range queries on the fast path, thus avoiding any contention on the RQC. Therefore, the fast-only and two-path skip hashes are able to outperform all other data structures by a significant margin. However, the slow-only skip hash suffers from contention on the RQC, leading to performance degradation. Allowing range queries to reuse version numbers would not help much, as they would still contend when inserting into `range_ops`. This behavior is a consequence of the small range size. We study it further in Section V-B3.

2) Mixed Workloads: The main difficulty with range queries is not running them in isolation—which can be done without any special mechanism—but keeping them both linearizable and performant in the face of concurrent updates. Therefore, in Figures 5d–5f, we evaluate workloads that do both. As before, range queries access 50 elements on average.

For each workload in this section, worker threads choose what type of operation to do at random, based on a distribution specified by that workload. For this reason, it does not make sense to separate throughput by operation type.

First, Figure 5d shows throughput as thread count is varied for a workload with a 10% updates and 10% range queries. The fast-only and two-path skip hashes maintain a moderate lead over the vCAS BST and a $> 2\times$ speedup over all other data structures. The difference in performance results from a small fraction of range queries needing more than 3 attempts to complete as a single STM transaction. In the fast-only algorithm, these additional attempts are made and succeed. In the two-path algorithm, the slow path is chosen instead, resulting in higher latency.

In Figure 5e, each thread performs 80% update operations and 20% range queries. Since each range query counts as

one operation, all numbers are significantly lower than in the previous chart. However, the general trends are otherwise the same: Increasing the update ratio does not impact fast-path range queries, but slow-path range queries continue to suffer. We profiled this overhead, and found that it was a result of two factors. First, slow-path range queries were competing to attain version numbers. Second, removal operations were aborting due to contention as they tried to delegate unstitching to in-flight range queries.

Lastly, in Figure 5f, we consider a workload with a much greater proportion of update operations per range query (98 instead of 4). This change greatly increases the probability a given range query will be aborted by a concurrent update, but reduces contention on global metadata on the slow path. The slow-path throughput improves significantly but not enough to be competitive. Apart from that, all trends are the same. We found that remove operations are not as common a source of aborts, but the slow-path range queries were still competing to acquire version numbers. In short, we conclude that our RQC mechanism is a bottleneck for short-running range queries due to interactions with concurrent range queries, not concurrent removals. However, the two-path approach all but eliminates the need for a slow path for this kind of workload.

3) Varying Range Query Size: From the above discussion, it may seem that there is no utility in having a slow path. Recall, however, that starvation is a well-known problem for long-running read-only transactions [48], and in the previous section, range queries were short. To understand how range query length affects performance of the evaluated maps, we conducted an experiment in which we varied that parameter rather than the thread count. Thread count was held constant at 48, split evenly into 24 update-only threads and 24 range-only threads, with all threads on one socket.

Figure 6 shows the results of this experiment. Since different operation types are performed by different threads, their throughput can vary independently, so the figure consists of two charts. The upper chart depicts update throughput in millions of operations per second, as before. The lower chart measures the performance of range queries but, instead of counting the number of range queries performed, it shows *how many key/value pairs were processed*. This improves readability at large range lengths.

The throughput of update operations remains almost constant across range lengths for most maps. For the skip hashes with fast-only and two-path range queries, their performance remains consistent and significantly faster than all data structures. The only surprise in this figure is for the slow-only skip hash. Here, we observe that elemental throughput improves for ranges of length $\geq 2^9$, eventually matching the peak throughput achieved by the other skip hashes.

The chart showing range throughput, on the other hand, is not flat. All evaluated maps improve in performance from 2^4 to 2^{10}, because they amortize the boundary overheads of a range query over a greater number of elements accessed. That is, all evaluated maps have some overhead involved in setting up and/or tearing down a range query. As range queries become longer, they are able to spend a proportionately greater time processing nodes. The performance of prior work plateaus once the length of the range queries becomes large enough to make this overhead irrelevant. However, the performance of the fast-path skip hash degrades rapidly for range queries starting at 2^{11}. This is after the point at which slow-path range query performance improved, and so the two-path skip hash is able to use the slow-path after three failed fast-path attempts.

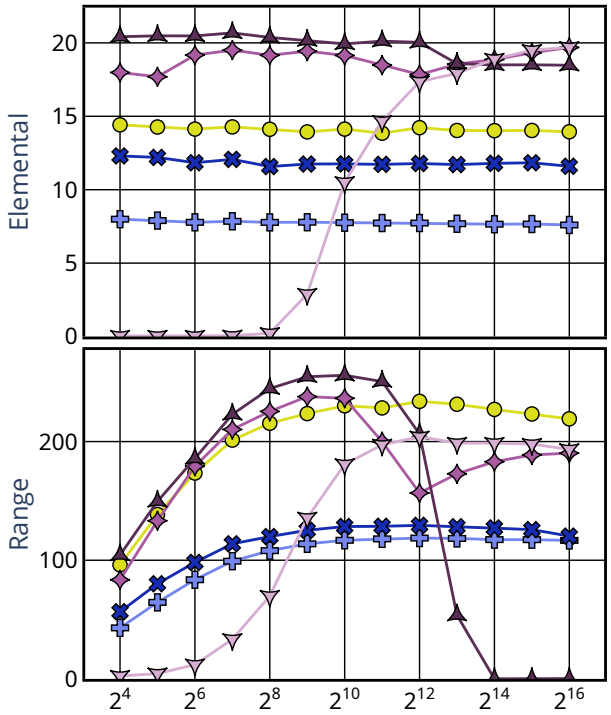


Fig. 6: Experiments with 24 update-only and 24 range-only threads as range query length varies. Top: update throughput (in millions of ops/sec). Bottom: range query throughput (in millions of nodes processed/sec).

Range Length	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}
Abort Rate	1.07	1.34	2.66	22.8	∞

TABLE I: Aborts per successful range query in a fast-path-only skip hash, by range length.

Table I reports the average number of aborts per successful range query as a function of the range query size in the fast-only variant. We see a precipitous increase in aborts as the query size increases until, at a range size of 2^{14}, no range query is able to complete. These aborts require some explanation. Recall that read-only STM transactions only abort if (1) they encounter an element that was updated after they began, and (2) they cannot prove that everything they read before that encounter is unchanged. In our highly optimized implementation of read-only transactions, there is no support for disproving condition (2). So then, when a fast-path range transaction encounters a modification that occurred after it began, it cannot prove that it can linearize, so it aborts. Were we to enable condition (2), latency would increase for

all range queries, but in a uniform workload the expected impact would only be a factor of two improvement in range size before the same abort frequency occurs. In short, long-running fast-path range queries do not have an easy way to avoid starvation. Fortunately, even our crude mechanism for transitioning to the slow path alleviates most of this overhead. Note that this experiment motivates transitioning to the slow path earlier, while previous experiments favored moving to the slow path later. Should practitioners choose to use the skip hash, we recommend they explore more nuanced strategies for switching to the slow path.

VI. Future Work

In the short term, we plan to study whether a hardware clock could avoid contention on RQC metadata. We hypothesize that integrating techniques developed by Grimes et al. [49] into the RQC could eliminate the contention we observed for short slow-path range queries. As a second direction, since our work has shown that low-level STM implementation details do matter, there may be yet-undiscovered optimizations specific to STM-based data structure design.

In the medium term, we intend to study better heuristics for transitioning range queries from the fast path to the slow path. Our current technique does not leverage any knowledge of the workload, nor does it allow the programmer to provide any input. In our microbenchmarks, it is trivial to craft policies that track with the best skip hash variant at any range size by leveraging such information. However, we chose not to present such configurations, since we are not yet convinced that such policies would be easy to create for real-world workloads. Further study is needed.

Another important research opportunity relates to organizing data to increase locality for range queries. Recently, Blelloch and Wei introduced an extension to vCAS that supports storing many key/value pairs in each node (i.e., a B-Tree) [53]. This can improve long-running range queries, by reducing cache misses. Similar locality-improving techniques have not been explored for STM data structures. Our initial analysis has concluded that the skip hash could be adapted to achieve similar improvements in range query performance, but more work is needed if the technique is to generalize to a broader class of STM-based data structures.

Lastly, we recognize that the skip hash's asymptotic complexity and low number of memory accesses per operation make it an appealing candidate for distributed and persistent memory systems. These systems have higher average memory access latencies than traditional shared memory multiprocessors, so reducing the number of memory accesses is essential to achieving good performance. This will, of course, necessitate a renewed focus on distributed and/or persistent STM, particularly one that incorporates assumptions analogous to those that resulted in the fast STM systems that facilitated this research.

VII. Conclusions

In this paper, we introduced the skip hash, which uses modern software transactional memory (STM) to compose a closed-addressing hash map with a doubly linked skip list. The resulting data structure has $O(1)$ complexity for most operations, resulting in exceptional throughput for elemental and point-query methods. By virtue of its use of STM, the skip hash easily supports linearizable range queries, but these risk starvation for high-contention workloads. To overcome this problem, we introduced a lightweight versioning technique that allows linearizable range queries to ignore values added after they linearized, and that also defers reclamation of logically deleted nodes that they may still need.

In experimental evaluation, we saw that the skip hash outperformed the state of the art in almost every workload configuration, and often by a large margin. We also showed that our fast-path/slow-path mechanism is effective in preventing starvation for long-running range queries.

The skip hash design is unconventional, both in its use of double-linking and in the way it composes data structures. However, its implementation is not complicated: the entire data structure requires under 1000 lines of code. These properties are a direct consequence of our decision to treat fast STM as a given, and assume that atomic multi-word transactions would "just work". In short, using STM simplified our design, implementation, and verification tasks.

References

[1] M. P. Herlihy and J. M. Wing, "Linearizability: a Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[2] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

[3] T. Brown, W. Sigouin, and D. Alistarh, "Pathcas: an efficient middle ground for concurrent search data structures," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 385–399.

[4] S. Timnat, M. Herlihy, and E. Petrank, "A Practical Transactional Memory Interface," in *Proceedings of the 2015 Euro-Par Conference*, Vienna, Austria, Aug. 2015.

[5] R. Guerraoui and V. Trigonakis, "Optimistic Concurrency with OPTIK," in *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming*, Feb. 2016.

[6] Y. Sheng, A. Hassan, and M. Spear, "Separating Mechanism from Policy in STM," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, Vienna, Austria, Oct. 2023.

[7] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[8] A. Kogan and E. Petrank, "A Methodology for Creating Fast Wait-Free Data Structures," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, Feb. 2012.

[9] T. Brown and H. Avni, "Range queries in non-blocking k-ary search trees," in *International Conference On Principles Of Distributed Systems*. Springer, 2012, pp. 31–45.

[10] M. Arbel-Raviv and T. Brown, "Harnessing Epoch-Based Reclamation for Efficient Range Queries," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vienna, Austria, Feb. 2018.

[11] Y. Wei, N. Ben-David, G. E. Blelloch, P. Fatourou, E. Ruppert, and Y. Sun, "Constant-time snapshots with applications to concurrent data structures," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 31–46.

[12] d. Miller, A. Hassan, and R. Palmieri, "Brief announcement: Lit: Lookup interlocked table for range queries," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, 2024, pp. 69–71.

[13] J. Nelson-Slivon, A. Hassan, and R. Palmieri, "Bundling linked data structures for linearizable range queries," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 368–384.

[14] W. Zhong, C. Chen, X. Wu, and S. Jiang, "{REMIX}: Efficient range query for {LSM-trees}," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 51–64.

[15] D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy, "Kiwi: A key-value map for scalable real-time analytics," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 357–369.

[16] I. Kokorin, V. Yudov, V. Aksenov, and D. Alistarh, "Wait-free trees with asymptotically-efficient range queries," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 169–179.

[17] P. Fatourou and E. Ruppert, "Lock-free augmented trees," *arXiv preprint arXiv:2405.10506*, 2024.

[18] G. Sela and E. Petrank, "Concurrent aggregate queries," *arXiv preprint arXiv:2405.07434*, 2024.

[19] H. Avni, N. Shavit, and A. Suissa, "Leaplist: Lessons Learned in Designing TM-Supported Range Queries," in *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Jul. 2013.

[20] R. Oshman and N. Shavit, "The skiptrie: low-depth concurrent search without rebalancing," in *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, 2013, pp. 23–32.

[21] M. Areias and R. Rocha, "On extending a fixed size, persistent and lock-free hash map design to store sorted keys," in *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*. IEEE, 2018, pp. 415–422.

[22] A. Prokopec, N. Bronson, P. Bagwell, and M. Odersky, "Concurrent Tries with Efficient Non-Blocking Snapshots," in *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*, Feb. 2012.

[23] A. Prokopec, "Cache-tries: concurrent lock-free hash tries with constant-time operations," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 137–151.

[24] M. P. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[25] N. Shavit and D. Touitou, "Software Transactional Memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, ON, Canada, Aug. 1995.

[26] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian, "Design and Implementation of Transactional Constructs for C/C++," in *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[27] H.-J. Boehm, "Transactional Memory Should Be an Implementation Technique, Not a Programming Interface," in *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, Mar. 2009.

[28] M. Abadi, A. Birrell, T. Harris, and M. Isard, "Semantics of Transactional Memory and Automatic Mutual Exclusion," in *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, San Francisco, CA, Jan. 2008.

[29] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc, "Practical Weak-Atomicity Semantics for Java STM," in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, Jun. 2008.

[30] A. Dragojevic and T. Harris, "STM in the Small: Trading Generality for Performance in Software Transactional Memory," in *Proceedings of the EuroSys2012 Conference*, Bern, Switzerland, Apr. 2012.

[31] N. Ben-David, G. E. Blelloch, and Y. Wei, "Lock-free locks revisited," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 278–293.

[32] P. Ramalhete and A. Correia, "Scaling up transactions with slower clocks," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2024, pp. 2–16.

[33] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sep. 2006.

[34] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime," in *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[35] Y. Lev, V. Luchangco, and M. Olszewsk, "Scalable Reader-Writer Locks," in *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[36] W. Ruan, Y. Liu, and M. Spear, "Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 4, pp. 40:1–40:21, Dec. 2013.

[37] D. Dice and N. Shavit, "Understanding Tradeoffs in Software Transactional Memory," in *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, San Jose, CA, Mar. 2007.

[38] P. Felber, C. Fetzer, and T. Riegel, "Dynamic Performance Tuning of Word-Based Software Transactional Memory," in *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[39] P. Ramalhete and A. Correia, "Scaling Up Transactions with Slower Clocks," in *Proceedings of the 29th ACM Symposium on Principles and Practice of Parallel Programming*, Edinburgh, UK, Mar. 2024.

[40] T. Riegel, C. Fetzer, and P. Felber, "Automatic Data Partitioning in Software Transactional Memories," in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, Jun. 2008.

[41] A. Dragojevic, Y. Ni, and A.-R. Adl-Tabatabai, "Optimizing Transactions for Captured Memory," in *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[42] T. Harris, M. Plesko, A. Shinar, and D. Tarditi, "Optimizing Memory Transactions," in *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*, Ottawa, ON, Canada, Jun. 2006.

[43] N. Ben-David, G. Blelloch, and Y. Wei, "Lock-Free Locks Revisited," in *Proceedings of the 27th ACM Symposium on Principles and Practice of Parallel Programming*, Seoul, South Korea, Feb. 2022.

[44] K. Fraser, "Practical Lock-Freedom," Ph.D. dissertation, King's College, University of Cambridge, Sep. 2003.

[45] A. Srivastava and T. Brown, "Elimination (a, b)-trees with fast, durable updates," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 416–430.

[46] T. Brown, A. Prokopec, and D. Alistarh, "Non-blocking interpolation search trees with doubly-logarithmic running time," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 276–291.

[47] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2014, pp. 317–328.

[48] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance Pathologies in Hardware Transactional Memory," in *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, Jun. 2007.

[49] O. Grimes, J. Nelson-Slivon, A. Hassan, and R. Palmieri, "Opportunities and limitations of hardware timestamps in concurrent data structures," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 624–634.

[50] Supplementary material. [Online]. Available: https://bit.ly/ICDCS2025-508

[51] J. Evans, "jemalloc memory allocator," 2017, http://jemalloc.net/.

[52] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[53] G. E. Blelloch and Y. Wei, "Verlib: Concurrent versioned pointers," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2024, pp. 200–214.