Exploiting Locality in Scalable Ordered Maps

Matthew Rodriguez
Lehigh University
mar316@lehigh.edu

Ahmed Hassan
Lehigh University
ahh319@lehigh.edu

Michael Spear
Lehigh University
spear@lehigh.edu

***Abstract*—We present the skip vector, a novel high-performance concurrent data structure based on the skip list. The key innovation in the skip vector is to flatten the index and data layers of the skip list into vectors. This increases spatial locality, reduces synchronization overhead, and avoids much of the costly pointer chasing that skip lists incur.**

We evaluate a skip vector implementation in C++. Our implementation coordinates interactions among threads by utilizing optimistic traversal with sequence locks. To ensure memory safety, it employs hazard pointers; this leads to tight bounds on wasted space, but due to the skip vector design, does not lead to high overhead. Performance of the skip vector for small data set sizes is higher than for a comparable skip list, and as the amount of data increases, the benefits of the skip vector over a skip list increase.

***Index Terms*—Concurrency, synchronization, linearizability, speculation, concurrent data structures, ordered maps**

I. Introduction

Spatial locality is one of the most important factors when optimizing the performance of software. Cache misses are expensive, and thus it is advantageous to organize data such that they can be brought to the processor with as few misses as possible. Indeed, the constant factors associated with poor locality can be so significant that they overshadow asymptotic complexity for some workloads. This point was articulated by Stroustroup in 2012 [1]. We repeat his experiments on a modern Intel Xeon Platinum CPU in Figure 1.

The figure shows a single-threaded set microbenchmark. The workload mix is 80% `Lookup`, 10% `Insert`, 10% `Remove`, with keys drawn from a uniform distribution whose range is given on the X axis. The set is pre-populated with half the keys in its range, so that the set size is stable throughout the experiment. We compare four set implementations: an unsorted vector ($O(n)$ overhead), a sorted vector ($O(n)$ `Insert` and `Remove`, $O(lg(n))$ `Lookup`), a C++ map (balanced internal tree with $O(lg(n))$ overhead), and a skip list [2] ($O(lg(n))$ overhead). The unsorted vector has the worst asymptotic behavior, but gives the best performance up to 7-bit key ranges. The sorted vector outperforms the map until 8-bit key ranges, and the skiplist up to 14-bit key ranges. The precipitous drops in performance for the vectors as the key range increases establish that there is a tradeoff between locality and asymptotic complexity: For large data sets, asymptotic complexity becomes more important.

For large data sets, it is important for the data structure to allow concurrent access. This introduces a tradeoff between spatial locality and scalability. When concurrent operations access data that are logically disjoint, but those data are on the same cache line, then performance can degrade due to false sharing and increased contention. The skip list, which has the worst locality in Figure 1, is least prone to these problems, and hence can be engineered to provide exceptional scalability. Consequently, skip lists have been used as the foundation for highly scalable ordered maps and sets [3], as well as strict [4] and relaxed [5] priority queues.

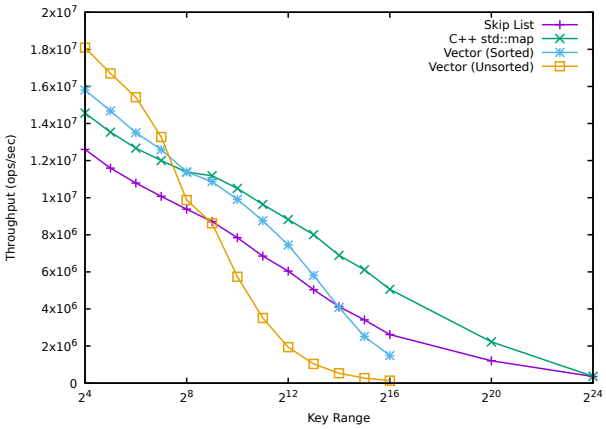


Fig. 1: Sequential set performance as a function of key range, for an 80/10/10 contains/insert/remove operation mix.

Many concurrent data structures are based on trees, but skip lists possess several advantages over trees. In particular, verification is easier, and the number of contention hotspots is lower. Furthermore, rebalancing a tree in a concurrent environment is tricky and poses many synchronization challenges; however, skip lists never need to be rebalanced, and any portion of a skip list operation that maintains the skip list's structure can be performed off of the critical path [6]. While many applications can use a hash map, applications that rely upon the order among keys cannot. Indeed, when ordered traversal (i.e., range queries) is an important part of an application, skip lists tend to offer the best performance. Furthermore, when an application thread performs a mutating range query on an ordered map *concurrently with* insertions and removals by other threads, skip lists can provide strong guarantees (i.e., linearizability [7]) about the values modified by the range query [8].

Figure 2 presents the general shape and structure of a skip list. The skip list is defined as a sorted linked list (the data layer), upon which increasingly sparse sorted "index" lists are layered. In the figure, the data layer contains $\{1, 5, 30, 33, 36,$

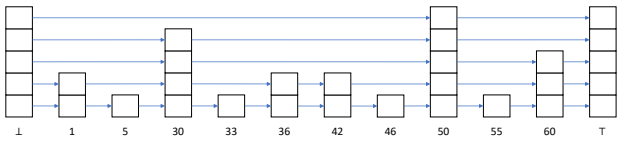


Fig. 2: A skip list set

$42, 46, 50, 55, 60\}$, the first index layer contains $\{1, 30, 36, 42, 50, 60\}$, and so forth. The "height" of a node is determined at insertion time, according to a geometric distribution, so that (w.h.p.) each index layer is half the size of the next lowest layer. These index layers allow a thread to "skip" forward when traversing the list. For example, consider a `Lookup` searching for key 55. The first traversal, in the topmost layer, will skip immediately to key 50, without considering half the nodes in the data layer. When keys range from $0 \ldots k$, the skip list will have $lg(k)$ layers, and the asymptotic complexity of each operation will be $O(lg(k))$ with high probability.

It is possible to increase spatial locality by storing $K > 1$ elements in a node, a technique hereafter referred to as "chunking". The motivation for chunking in the data layer is obvious: under standard assumptions and a large key range, two operations will rarely operate on the same region of the data layer, so optimizations that increase locality should not increase contention. The simple but powerful observation that motivates our work is that the geometric distribution of node heights results in a complementary property in the index layers: for level h of the skip list, modifications are exceedingly rare (probability 2^{-h}). We posit that most of the nuance of nonblocking concurrent skip lists is hence unnecessary: At high layers, nodes are effectively immutable; at low layers, concurrent access is rare; and in between, these two properties change in inverse proportion. Thus chunking should be advantageous at all levels.

To evaluate this claim, we introduce a new data structure, the skip vector. The skip vector pushes spatial locality in skip lists to the extreme, resulting in the following benefits:

- It employs chunking at every level, so that each layer can be traversed with fewer cache misses.
- It needs logarithmically fewer levels than the corresponding skip list, resulting in less pointer chasing.
- Since (w.h.p.) operations access $lg(lg(n))$ nodes, the skip vector can use precise memory reclamation without incurring the high overheads typically associated with hazard pointers [9]–[11].
- The skip vector uses a novel extension of sequence locks [12] to allow reads to progress speculatively, and hence avoid contention.
- It offers a variety of options for tuning the size and implementation of its chunks, which is especially beneficial at large data set sizes.
- It trivially supports linearizable range operations.

The remainder of this paper is organized as follows. Section II discusses related research into scalable skip lists. Sections III and IV introduce the skip vector algorithm and argue its correctness. Section V presents a detailed evaluation. Finally, Section VI concludes and suggests future work.

II. Related Work

There are many scalable (un-chunked) skip lists [6], [13]–[16], which employ a common set of techniques. First, they treat nodes in each layer as immutable, and replace them instead of updating them (analogous to copy-on-write). Second, they leverage speculation and some form of deferred memory reclamation [16]–[18], so that traversing the index layer does not require locking or reference counting of nodes. Third, they postpone updates to the index layer (required for certain `Inserts` and `Removes`) until after the data layer is modified. Fourth, they mitigate the impact of concurrent updates by aggressively pursuing nonblocking progress guarantees. In particular, updates to the index layer are performed in a way that ensures that concurrent operations remain correct and need not block, even when the index layer is stale.

In addition, there are three significant chunked skip list designs. The LeapList [19] allows data layer nodes to hold up to K keys, and is synchronized by a combination of Software Transactional Memory [20] and consistency-oblivious programming [21]. In comparison to a non-blocking skip list, the LeapList showed slightly faster lookups and slightly slower inserts and removes. The LeapList can serve as either a set or a map. Its main benefit is that it provides linearizable read-only range queries that are an order of magnitude faster than the non-linearizable range queries of the baseline.

The concurrent unrolled skip list (CUSL) [22] also employs chunking in the data layer. However, its focus is on achieving nonblocking progress, not fast range queries. CUSL uses a novel group mutual exclusion technique to allow concurrent operations on a (chunked) data layer node. Unfortunately, the CUSL cannot serve as a map, only as a set. Additionally, the need for atomic update of elements under group mutual exclusion limits the types of keys: on a 64-bit machine, they can be no more complex than a 32-bit integer.

The GPU-Friendly Skip List (GFSL) [23] exploits memory coalescing within a GPU by using chunking in the index and data layers. The chunk size is tightly coupled to the GPU geometry, with each chunk having a maximum size that matches the number of threads in a warp. By using intra-warp collective operations (e.g., `_shfl`), the GFSL can process a chunk in constant time. Chunks are protected by locks, but since all fields of a chunk can be updated simultaneously by a full warp, locks are only needed for insertion and removal; lookup operations are lock free. Like the other chunked skip lists, and indeed all of the skip lists discussed in this section, precise memory reclamation is not possible for the GFSL: nodes used by the GFSL cannot be returned to the system until the GFSL reaches a quiescent state.

A. Summary and Design Goals

In the evolution from classic nonblocking skip lists to chunked skip lists, we observe the following challenges:

Listing 1: The skip vector map and its data types.

```
Type SkipVectorMap⟨KeyType, ValueType⟩
  layerCount            :  Integer
  targetIndexVectorSize :  Integer
  targetDataVectorSize  :  Integer
  mergeThreshold        :  Integer
  head                  :  Node*
Type SequenceLock
  sequenceNumber  :  Bit[61]
  isLocked        :  Bit
  isOrphan        :  Bit
  isFrozen        :  Bit
Type IndexNode extends Node
  vector  :  VectorMap⟨KeyType, Node*⟩
  next    :  IndexNode*
  lock    :  SequenceLock
Type DataNode extends Node
  vector  :  VectorMap⟨KeyType, ValueType⟩
  next    :  DataNode*
  lock    :  SequenceLock
Type VectorMap⟨KeyType, ValueType⟩
  size        :  Integer
  targetSize  :  Integer
  keys        :  KeyType[2 × targetSize]
  vals        :  ValueType[2 × targetSize]
```

1) Scalability appears to be at odds with precise memory reclamation, with all known scalable skip lists accepting significant worst-case space overheads.
2) Despite the appeal of chunking in index layers, only one algorithm has exploited it thus far, and could only do so by relying on special intra-warp GPU instructions.

We also observe the following opportunities:

1) Despite using locks in some cases, the LeapList and GFSL are able to scale on par with (or better than) their nonblocking competitors.
2) Chunking reduces the number of index layers, which reduces pointer chasing.
3) Flexibility in the size of chunks, and their implementation (especially sorted versus unsorted) have the potential to deliver new performance opportunities.

We now present our skip vector algorithm, which aims to address these challenges and exploit these opportunities.

III. Design of a Concurrent Skip Vector Map

The skip vector retains the asymptotic guarantees and scalability of the skip list, while increasing locality and reducing manual memory management overheads. Structurally, a skip vector is similar to a skip list, consisting of nodes organized into layers. The primary difference is that sequences of adjacent layers are flattened into vectors. Listing 1 presents the data types for the skip vector map.

An example skip vector is depicted in Figure 3. It is initialized with two nodes in each layer containing sentinel keys $\perp$ and $\top$ which serve a purpose similar to head and tail nodes in a skip list (Figure 3a). The skip vector keeps a pointer to head, the head node in the top index layer. The layers are indexed from bottom to top starting with the data layer as layer 0. The keys *within* a node may or may not be sorted (see Section V), but keys are always ordered *among* nodes.

The data layer is an ordered set of all key-value pairs in the map, implemented as a list of DataNodes. Index layers are ordered sets of key-pointer pairs, implemented as lists of IndexNodes. For each key-pointer pair $\langle K, \mathsf{down}\rangle$, down points to a node in the layer beneath whose first element is K.

Nodes contain a VectorMap, a map implemented using two fixed-capacity vectors for keys and values correlated by index. size is the current number of elements. The parameter targetSize (or T) determines the expected size of each node; we use $2T$ as the capacity. When a node n exceeds capacity, we create a new node o, move the latter half of n's elements to o, and insert o immediately after n. Note that there is no pointer to o in the layer above; it can only be reached by following n's next pointer. We call nodes with this property *orphan* nodes, as they lack a *parent* node in the layer above.

Each node has a SequenceLock, implemented as a 64-bit integer, from which we re-purpose two bits as flags: isOrphan and isFrozen. The former indicates if the node is an orphan. The remaining 62 bits behave as an ordinary sequence lock: the least significant bit, isLocked, indicates if the lock is held, and the remaining bits form the sequenceNumber. (isFrozen will be discussed in Section III-B.)

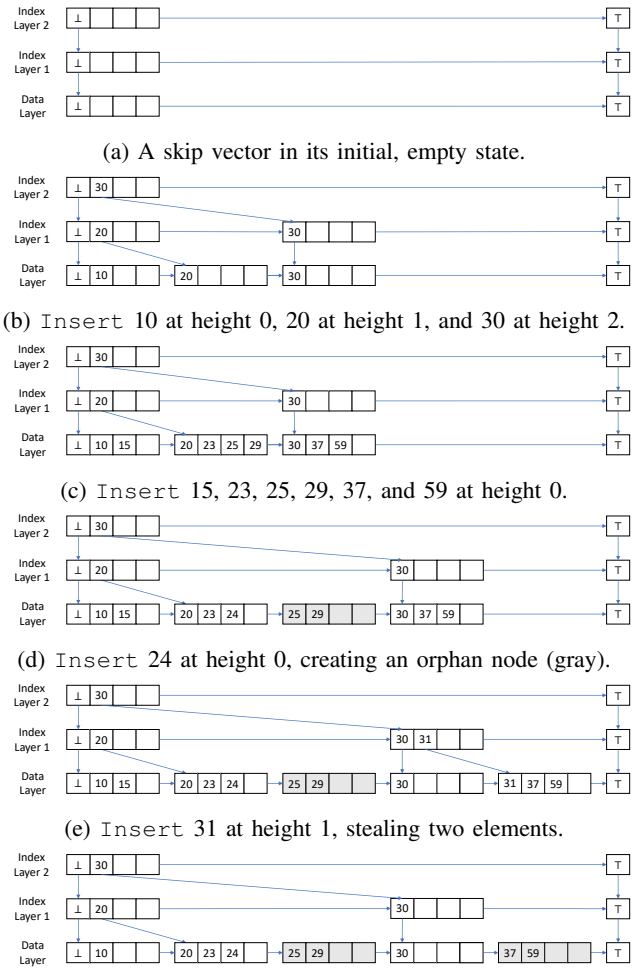
Figure 3 illustrates the behavior of a skip vector. First, three keys are inserted at random heights (Figure 3b). Each is placed into the data layer, but the taller keys are also placed into the index layers. In Figure 3c, several more keys are inserted at height 0. Figure 3d inserts key 24, but since the destination data node is full, it is split, creating an *orphan*, shaded gray. Figure 3e's insertion of 31 at height 1 requires splitting an existing data node, stealing all elements > 31.

Orphans can also be created by removals. Figure 3f shows the result of removing 31 from the data structure. Removing 31 from the index layer removes a link to the rightmost data node, making it an orphan. This node should be merged into its predecessor, but to improve concurrency, merging is lazy: it is left for some future operation to do. Finally, suppose an attempt is made to insert key 59. It will fail, as 59 is already present, but the insert method will merge the orphan, resulting in the skip vector reverting to Figure 3d.

A. The Sequential Algorithm

We now describe a sequential implementation of the skip vector map. In our presentation, we refer to the listings for the concurrent algorithm, Listings 2, 3, and 4. The sequential algorithm is not listed; however, it is the same as the concurrent algorithm, only without locks or hazard pointers.

1) Lookup: Listing 2 presents the `Lookup` operation. The sequential algorithm does not need to use hazard pointers (lines 2, 13, 15, 19, 23, 38, 41), and can reclaim memory immediately (31). In addition, it does not need to use sequence locks (lines 3, 12, 16, 17, 18, 24, 25, 27, 28, 37, 40). `Lookup` searches for key K in a manner similar to a `Lookup` in a skip list: starting with head (the first IndexNode in the topmost layer), it searches rightward (via `TraverseRight`) for the node storing the largest key $\leq K$. Once found, it uses the corresponding pointer to move down one layer



(a) A skip vector in its initial, empty state.

(b) `Insert` 10 at height 0, 20 at height 1, and 30 at height 2.

(c) `Insert` 15, 23, 25, 29, 37, and 59 at height 0.

(d) `Insert` 24 at height 0, creating an orphan node (gray).

(e) `Insert` 31 at height 1, stealing two elements.

(f) `Remove` 31, causing a data node to become an orphan.

Fig. 3: Operations on a very small example skip vector.

(`ExchangeDown`). Each time the search moves downward, it has used the previous layer to skip into the new layer at a node whose minimum key is necessarily $\leq K$, and w.h.p. close to it. When the data layer is reached, w.h.p. K is within the range of keys stored in the chosen DataNode; if not, then K will be within the range of one of the DataNode's successors.

In more detail, suppose a `Lookup` at node n searches the keys in n.vector. There are two possibilities. In the common case, it finds a pair (K_f, down) (line 6), and K_f is not the largest key in n.vector. Then, the search can use down to move down one layer. In the second case, K_f is the last key in n.vector, and so the search must consider the node at n.next. If next's minimum key is $\leq K$, the search continues there.

It is advantageous for lookup operations to avoid modifying the data structure. However, as we will see, a `Remove` can leave an empty node, or a consecutive pair of nodes n_i, n_j where the count of elements in the two nodes is less than mergeThreshold. In the former case, a `Lookup` will remove the empty node. In the latter, a `Lookup` will overlook the violated invariant, but an `Insert` or `Remove` will repair it during its call to `TraverseRight` (line 26).

Listing 2: Pseudocode for thread-safe lookup in a concurrent skip vector with hazard pointers.

```
   function Lookup(K_k)
1      curr ← head
2      HP.take curr
3      curr_ver ← read curr.lock
       // Traverse rightward and downward to reach data layer
4      while typeof curr = IndexNode do
5          ⟨curr, curr_ver⟩ ← TraverseRight(curr, curr_ver, K_k)
6          ⟨K_f, down⟩ ← k/v pair for largest key ≤ K_k in curr.vector
7          ⟨curr, curr_ver⟩ ← ExchangeDown(curr, curr_ver, down)
       // Traverse rightward in data layer
8      ⟨curr, curr_ver⟩ ← TraverseRight(curr, curr_ver, K_k)
9      if curr.vector.Contains(K_k) then
10         result ← curr.vector.GetValue(K_k)
11     else result ← ⊥
12     verify curr.lock = curr_ver else HP.dropAll restart
13     HP.dropAll
14     return result
   // Move downward via hand-over-hand sequence(read) locking
   function ExchangeDown(curr, curr_ver, newNode)
15     HP.take newNode
16     verify curr.lock = curr_ver else HP.dropAll restart
17     new_ver ← read newNode.lock
18     verify curr.lock = curr_ver else HP.dropAll restart
19     HP.drop curr
20     return ⟨newNode, new_ver⟩
   // Move rightward via hand-over-hand sequence(read) locking
   // – may perform cleanup of empty/underfull nodes in any layer
   function TraverseRight(curr, curr_ver, K_k)
21     next ← curr.next
22     while curr.size = 0 ∨ K_k > curr.maximumKey do
23         HP.take next
24         verify curr.lock = curr_ver else HP.dropAll restart
25         next_ver ← read next.lock
           // uncommon case: merge/remove nodes left by prior Remove() calls
26         if next.isOrphan ∧ (next.size = 0 ∨ (called by Insert or Remove ∧ curr.size + next.size < mergeThreshold)) then
27             tryUpgrade curr.lock else HP.dropAll restart
28             tryUpgrade next.lock else release curr.lock HP.dropAll restart
29             move all elements from next to curr
30             curr.next ← next.next
31             HP.mark next
32             release next.lock
33             next ← curr.next
34             curr_ver ← release curr.lock
35             continue
36         if K_k < next.minimumKey then
37             verify next.lock = next_ver else HP.dropAll restart
38             HP.drop next
39             break
40         verify curr.lock = curr_ver else HP.dropAll restart
41         HP.drop curr
42         ⟨curr, curr_ver⟩ ← ⟨next, next_ver⟩
43         next ← curr.next
44     return ⟨curr, curr_ver⟩
```

2) Insert: Listing 3 presents the pseudocode for an `Insert` operation. As with `Lookup`, the sequential version

Listing 3: Pseudocode for thread-safe insertion in a concurrent skip vector with hazard pointers.

```
   function Insert(K_k, V_v)
1      height ← randomly generated height
2      prevs ← []
3      layer ← layerCount − 1
4      curr ← head
5      HP.take curr
6      curr_ver ← read curr.lock
       // Traverse rightward and downward to reach data layer
7      while typeof curr = IndexNode do
8          ⟨curr, curr_ver⟩ ← TraverseRight(curr, curr_ver, K_k)
           // Will we add K_k at this index layer?
9          if layer ≤ height then
10             tryFreeze curr.lock else HP.dropAll restart
11             HP.drop curr // HP not needed for frozen nodes
12             prevs[layer] ← curr
13             set checkpoint // On a restart, resume from here
14         ⟨K_f, down⟩ ← k/v pair for largest key ≤ K_k in curr.vector
           // If K_k found in index, Insert() fails
15         if K_k = K_f then
16             verify curr.lock = curr_ver else HP.dropAll restart
17             thaw all frozen locks
18             HP.dropAll
19             return false
20         ⟨curr, curr_ver⟩ ← ExchangeDown(curr, curr_ver, down)
21         layer ← layer − 1
22     ⟨curr, curr_ver⟩ ← TraverseRight(curr, curr_ver, K_k)
23     tryFreeze curr.lock else HP.dropAll restart
24     prevs[0] ← curr
25     HP.drop curr
       // If K_k present, clean up and return
26     if curr.Contains (K_k) then thaw all frozen locks return false
       // If K_k absent, insert into appropriate layer(s)
27     value ← V_v
28     for layer ∈ [0, height − 1] do
29         acquire prevs[layer].lock // move node from frozen to locked
30         newNode ← new node(K_k, value)
31         move all elements > K_k from prevs[layer] to newNode
32         newNode.next ← prevs[layer].next
33         prevs[layer].next ← newNode
34         release prevs[layer].lock
35         value ← newNode
       // At the chosen insert height, K_k gets added to an existing node
36     acquire prevs[height].lock // move node from frozen to locked
37     prevs[height].Insert (K_k, value)
38     release prevs[height].lock
39     return true
```

of the algorithm would elide all uses of hazard pointers and locks, and also checkpointing (e.g. line 13).

The primary challenge in `Insert`, relative to `Lookup`, is that an `Insert` may need to update index layers. Such updates cannot be performed until after an operation has verified that the target key is absent in the data layer. An `Insert` of key *K* first generates a random height (line 1), using targetDataVectorSize (T_D) and targetIndexVectorSize (T_I). First, $\frac{T_D-1}{T_D}$ of insertions will have height 0, and thus only exist in the data layer. When height $= 0$, `Insert` proceeds much like a `Lookup`, except that once it reaches the desired data node n, it adds the mapping from *K* to V_v into n.vector (line 37). If n is full, then the operation splits n, creating an orphan to the right. For the remaining inserts, a height will be generated using geometric distribution with $p = \frac{1}{T_I}$ from 1 to layerCount $- 1$.

When height > 0, the operation searches for the appropriate data node as before, but along the way, it keeps track of the pointers to all of the index nodes it will need to modify in order to perform the `Insert` in an array prevs[] (line 12). A node will need to be modified if *K* is in its range and its height is $\leq$ height. Once the operation reaches the data layer at node n, it returns **false** if *K* is present (line 26). Otherwise, the operation inserts a new node into the data layer with *K* as its first element, and moves all of the elements $> K$ into it from n. It repeats a similar process for all index layers $<$ height, splitting the nodes at *K* and inserting a pointer to the newly inserted node below (lines 28–35). The process terminates at level height, where *K* is inserted into the appropriate node at that layer (line 37) without splitting it (unless it is at capacity). Finally, the operation returns **true**.

3) Remove: Listing 4 presents the `Remove` operation. The structure of the skip vector ensures that when *K* is present in index node I_l at level l, and I_l is not an orphan node, then either (a) *K* is the smallest element in I_l, and is also present in an index node at next highest level $l + 1$, or (b) *K* is not the smallest element in I_l. While we defer discussion of concurrency until Section III-B, the listing addresses two possible concurrent interleavings on lines 8 and 17. For the sequential code, these conditions cannot arise.

The main challenge that arises in this code relates to the removal of a key from the index layers. Otherwise, a `Remove` of key *K* proceeds much like a `Lookup`. In the common case where *K* is not observed in any index layer (lines 14–22), the result of the operation depends on whether *K* is found in the data layer, and no maintenance is required in index layers.

If *K* is found in any index layer, then the operation exits the index traversal early (line 12). It is guaranteed to find *K* as the first element in each subsequent index and data layer as it takes an optimized downward traversal path (lines 23–27). It removes *K* from every node where it was found (lines 24, 28), data layer and index layer alike, before returning **true**.

Note that for all but the top-most node, the removal of *K* will have the side effect of making that node an orphan (line 24). This can result in orphan nodes with too few elements, which should be merged with their predecessors. We explicitly offload this burden to future operations (`TraverseRight` lines 27–34). Whenever an `Insert` or `Remove` operation needs to check a next pointer, it also checks if the next node is an orphan, and whether the sum of the two nodes' sizes are less than mergeThreshold. If they are, then the orphan's elements are moved into its predecessor, and the orphan node is removed from the skip vector. When *any* operation discovers an empty orphan, it will remove it, as empty nodes violate the assumption that a minimum element exists (`TraverseRight` line 36).

Listing 4: Pseudocode for thread-safe removal in a concurrent skip vector with hazard pointers.

```
   function Remove(K_k)
1      curr ← head
2      HP.take curr
3      curr_ver ← read curr.lock
       // Traverse rightward and downward to reach data layer
4      while typeof curr = IndexNode do
5          ⟨curr, curr_ver⟩ ← TraverseRight(curr, curr_ver, K_k)
6          ⟨K_f, down⟩ ← k/v pair for largest key ≤ K_k in curr.vector
           // Do we need to remove from this index layer?
7          if K_k = K_f then
               // Handle concurrent interleaving.
8              if curr.minimumKey = K_k ∧ ¬curr.isOrphan then
9                  HP.dropAll restart
               // Subsequent layers can be traversed non-speculatively
10             tryUpgrade curr.lock else HP.dropAll restart
11             HP.drop curr
12             break
13         ⟨curr, curr_ver⟩ ← ExchangeDown(curr, curr_ver, down)
       // Common case: K_k not in any index layer
14     if typeof curr = DataNode then
           // Search rightward for key, remove it if found
15         ⟨curr, curr_ver⟩ ← TraverseRight(curr, curr_ver, K_k)
16         tryUpgrade curr.lock else HP.dropAll restart
17         if curr.minimumKey = K_k ∧ ¬curr.isOrphan then
18             HP.dropAll restart
19         result ← curr.Remove(K_k)
20         release curr.lock
21         HP.drop curr
22         return result
       // Upon break@line 12, process layers downward
23     while typeof curr = IndexNode do
24         down ← curr.Remove(K_k)
25         acquire down
26         down.isOrphan ← true
27         release curr.lock
28     curr.Remove(K_k)
29     release curr.lock
30     return true
```

B. Extensions for a Concurrent Skip Vector

This subsection describes the algorithmic modifications for concurrency; their sufficiency is argued in Section IV-C. We employ sequence locks [12] and hazard pointers [9] to make the skip vector concurrent. A sequence lock is a spinlock based on a strictly increasing counter. Even values indicate the lock is unheld, odd values indicate the lock is held, and a read-only critical section can operate speculatively by reading the lock, reading data, and then re-reading the lock to ensure it is both even and unchanged. We augment each index and data node by attaching a sequence lock to it. As discussed in Section III-A, we have re-purposed two bits from this sequence lock to represent boolean values, isOrphan and isFrozen. Note that the use of sequence locks introduces low-level code changes in order to remain compliant with the C++ memory model [12].

Threads use sequence locks to traverse the data structure optimistically. Operations employ a hand-over-hand strategy, read-locking each node visited. If an operation observes a change to a sequence lock that it has read-acquired, it aborts and retries (`Lookup` lines 12, 16, 18, 24, 27, 28, 37, 40). Write-locks are only acquired just before an operation modifies a node, and released immediately afterward.

To reduce blocking, `Insert` uses the isFrozen bit. Freezing a node puts it into a state where only the thread that froze it can acquire it, but other threads can still read it. In this way, an `Insert` may block conflicting `Insert`s and `Remove`s, without delaying the traversal of other operations.

Our discussion uses several keywords to encapsulate complex behavior. **restart** is used to restart an operation after a sequence lock read failure. If **restart** is invoked from inside one of the helper functions—`ExchangeDown` or `TraverseRight`—it is the top-level `Lookup`, `Insert`, or `Remove` that is restarted. `Insert` may invoke **restart** after acquiring a write-lock on a node; if it does, it restarts from the last node it locked rather than the start. This is indicated in the listing with the line "set checkpoint" (line 13).

After reading a node, threads use **verify** to ensure that the read was valid by checking the sequence lock again. If another thread changes the sequence number or sets isLocked, the operation will drop all held hazard pointers and invoke **restart**.

In languages with manual memory reclamation, threads take hazard pointers on nodes prior to accessing them, denoted as **HP.take**. There is a danger that the node was deleted concurrently with the acquisition of its hazard pointer. The combination of hazard pointers and sequence locks leads to a useful optimization: the thread must have found that pointer in another node, so its **verify** of that node also indicates that the hazard pointer was successfully taken. This optimization reduces instructions, and also avoids read-read memory fences on systems with relaxed memory consistency.

acquire and **release** take and release a sequence lock, respectively. **acquire** spins until isLocked bit is clear, then use a compare-and-swap operation in order to set the isLocked bit and therefore take the lock for this thread. **release** atomically clears isLocked and increments sequenceNumber. **tryUpgrade** is used when a thread has performed a **read** on a sequence lock, and wishes to upgrade itself to a writer on that node. This keyword combines a **verify** with an **acquire**, acquiring a sequence lock only if its value is unchanged, and triggering a **restart** otherwise. Finally, **tryFreeze** is similar to **tryUpgrade**, but merely tries to set isFrozen.

IV. Correctness

A. Sequential Correctness

First, we argue that the skip vector correctly implements a sequential map interface. We say that K is the set of keys in the map, and V is the set of values in the map, such that there is a unique $v_i \in V$ for each $k_i \in K$. We represent the mapping from k_i to v_i as $k_i \rightarrow v_i$. Initially, K and V are empty, and thus $\forall k_i, k_i \rightarrow \perp$. From an arbitrary state, the sequential specification of a map requires the following. First, if `Insert`(k', v') is invoked, and $k' \in K$, **false** is returned. Otherwise k' is added to K, v' is added to V, a mapping

$k' \to v'$ is created, and **true** is returned. Second, in response to `Lookup`(k'), if $k' \notin K$, then $\perp$ is returned. Otherwise there must be some mapping $k' \to v'$, and v' is returned. Third, when `Remove`(k', v') is invoked, if $k' \notin K$, then **false** is returned. Otherwise there must be a mapping $k' \to v'$. In this case, the mapping is removed, k' is removed from K, v' is removed from V, and **true** is returned.

The correctness argument for the sequential skip list is similar to the correctness argument of the skip list. When a skip list inserts a node, it uses a geometric distribution to determine the height. The distribution uses parameter p to indicates the relative probabilty between two adjacent values. Skip lists commonly use $p = 0.5$, so that each height is half as dense as the one below it. We also note that each index layer, as well as the data layer, can be implemented with any arbitrary map data structure, so long as it is possible to maintain pointers into the next lower layer. Each layer could, for example, be a single array, or a list of arrays. Therefore, we argue that a skip vector is a skip list where the layers are implemented as lists of arrays rather than linked lists, with $p = \frac{1}{T}$, where T is the targetSize of the index and data layers. (We assume for brevity and clarity that the index and data layers have the same value for targetSize, but the argument is similar when they do not.) The correctness of the skip vector therefore follows directly from the correctness of a skip list.

B. Asymptotic Guarantees

A skip list with n elements needs $log_{1/p}(n)$ layers to maintain asymptotic guarantees. (For example, when $p = \frac{1}{2}$ and $n = 2^{32}$, it needs $log_2(2^{32}) = 32$ layers.) The minimum number of layers a skip vector needs, then, is $log_T(n)$. Any operation on a vector of length T grows with respect to T. As T is a constant, this is $O(1)$. At each layer, we expect w.h.p. that the number of consecutive orphans is less than some constant c. Therefore, a `Lookup` spends constant time at each layer. With $log_T(n)$ layers, the total run time is proportional to $log_T(n)$, i.e. $O(log(n))$.

Following the above logic, so long as the amount of work done at each layer can be shown to be constant, then the whole operation is $O(log(n))$. Therefore, consider a a worst case `Insert`, which modifies all layers. In addition to the cost of traversing the data structure (constant per layer, like `Lookup`), an `Insert` would also have to perform split and insert operations on at most one vector per layer. Again, as T is constant, this is constant. Similar logic applies to a worst-case `Remove`. Up until now we have ignored the cost of merging orphans. Although an operation may merge many nodes per layer in a pathological case, each `Insert` creates at most one node per layer, and so the cost of merging can be amortized to the `Insert` that creates the node, which still amounts to constant work per layer for that `Insert`.

C. Concurrent Correctness

In this section, we argue that our concurrency scheme is linearizable [7] and deadlock-free. First, consider a simpler concurrency scheme in which each node has an ordinary mutex instead of a sequence lock, and locks are not released until the end of an operation. By replacing **read** operations with mutex acquires, we immediately achieve a race-free algorithm: the fields of a node are only accessed by a thread while it is holding the lock. Similarly, the algorithm is trivially deadlock-free, due to the absence of cycles in the lock graph: locks in higher levels are always acquired before locks in lower levels, and within a level, locks are always acquired in ascending order. For this simpler algorithm, the argument for either serializability or linearizability is also trivial, since it obeys two-phase locking [24]. Each operation can happen immediately after its last lock is acquired.

Next, consider an implementation that replaces the mutex lock on each node with a starvation-free readers/writer lock. Each operation now takes a read lock or a write lock on each node as needed. This implementation remains correct with regard to atomicity, as the result still obeys two-phase locking.

Third, let us transform the implementation to use sequence locks with a freeze bit in place of readers/writers locks. There are two new concerns. First, the skip vector must be designed so that it is possible to safely discard any invalid results that may be computed due to a reader seeing inconsistent state. In our algorithm, we accomplish this by having any operation that sees inconsistent state restart from either the beginning, or a checkpoint node known to be unchanged because the operation has it frozen. Second, all accesses must be memory-safe. We achieve this by taking a hazard pointer prior to any risky read. Additionally, the node's vectors have a fixed size, and all vector operations are implemented in a way that they are guaranteed to terminate even if concurrent changes occur. Therefore, there is no way to go out of an array's bounds or fail to reach the end of its reader critical section.

The final transformation is to do away with two-phase locking for the read-only part of an operation, and replace it with hand-over-hand locking, such that any read-only traversal phase needs at most two hazard pointers and two read locks at a time. It remains impossible for the nodes an operation currently has read-locked to change (without forcing a restart), so the only adverse effect this change creates is that it is now possible for an operation to end up in a situation where nodes it previously visited have changed without causing it to abort and retry. For this to be correct, we must show that either this operation fails and restarts, or it completes correctly—that is, it has the exact same effect as if it were canceled and restarted.

We observe that an operation on key *K* will never traverse through a node n unless n's minimum element is $\leq$ *K*. (`TraverseRight` may briefly visit the successor of such an n, but only to rule it out.) Thus, an operation can never "overshoot" the target element and end up too far to the right; it can only "undershoot," ending up too far to the left. Since the nodes of each layer form a list, if a `Lookup` undershoots in one level, `TraverseRight` will fix that in the next layer. Though it may take a longer path than it otherwise would, it will still reach the right data node and complete correctly.

The correctness of `Insert` immediately follows. An `Insert` of key *K* at height h will succeed if it first freezes the

correct node for *K* in layer h. This node will be reached in the same manner for both `Insert` and `Lookup`: overshooting is impossible, and undershooting in layer $L > h$ only results in more traversal in layer $L - 1$, where $L - 1 \geq h$. Once an `Insert` freezes its first node, its behavior is analogous to the mutex-based, two-phase locking case, but with freezing instead of acquiring locks. Once it has frozen all the way down to the data layer, it determines if the key is already present. If not, it upgrades frozen locks to acquired, updates the corresponding nodes, and releases. This behavior has the same serializability argument as two-phase locking, but since it updates nodes from the bottom up, it does not violate the correctness of the search phase of concurrent operations.

A concurrent `Remove` resembles hand-over-hand locking in a list: once *K* is found in node n_L at layer L, the node holding *K* one layer down (n_{L-1}) is locked, *K* is removed from n_L, and then the operation repeats at node n_{L-1}. For the highest level L such that $K \in n_L$, the act of reaching some node n_L is the same as for `Lookup`. The exceptional circumstance is that L must be the *highest* level at which *K* appears. Note that this problem is unique to `Remove`, and results from the fact that `Remove` does not know the height of a to-be-removed *K* until after its operation begins traversing the skip vector. In contrast, `Insert` pre-determines the uppermost height of its to-be-inserted *K* on line 1. If *K* is concurrently added to the skip vector by another `Insert`, then once the second `Insert` reaches a level that was also modified by the former `Insert`, it will return **false**.

Returning to `Remove`, if a concurrent `Insert` places *K* at height h, and `Remove` only removes *K* from levels $L < h$, then the skip vector will be malformed. When we consider the shape of the skip vector, an important invariant emerges. A down pointer for a given key *K* in an index node always points to a non-orphan node in the next layer down whose minimum key is *K*. Furthermore, we know that each key can only appear at most once in each layer. Therefore, if we find a key *K* in a node n in a layer L, we can determine whether *K* exists in any index layer above L by examining n. If n is an orphan node, or *K* is not the minimum key in n, or both, then *K* cannot appear in any layer above L. Therefore, when `Remove` first finds *K* in the data structure (line 7), it checks these two conditions (line 8). If either holds, then execution continues. Otherwise, the operation retries.

To conclude: through a series of refinements, we have established that our concurrency scheme is equivalent to two-phase locking. We have also argued that the lock graph is free of cycles, and thus deadlock-free. We also argued that data are never read without holding the corresponding node's lock (either as a writer or via a sequence lock read critical section). From the perspective of linearizability, successful `Insert` and `Remove` operations linearize on the instruction that write-acquires their last lock. All remaining operations are read-only, and linearize at the point where they find (or fail to find) their target key in some node of the data layer. The linearization point of these operations is the instruction that performs the final verification of the sequence lock at the data layer.

V. Evaluation

In this section, we evaluate the performance of the skip vector. Our main goal is to explore how well the skip vector scales on microbenchmarks and more realistic workloads. We also explore the impact of hazard pointers, the relative merit of chunking in the index and data layers, and the impact of skip vector configuration parameters.

All experiments were conducted on a machine with four Intel Xeon Platinum 8160 CPUs at 2.10GHz, which provided 96 cores/192 threads and 768GB of RAM. All data points are the average of five runs. We did not observe significant variance. Our machine ran Red Hat Enterprise Linux Server 7 with Linux kernel 3.10.0, and all code was compiled using g++ version 7.3.1 and O3 optimizations.

For our microbenchmark experiments, we (1) utilized 64-bit integers as the key type and 64-bit pointers as the value type, (2) pre-filled each data structure half of the keys in the range, in a NUMA-fair way, and (3) ran each trial for 5 seconds.

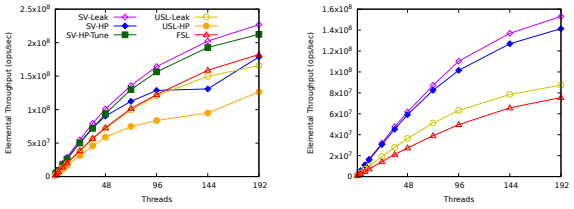
A. Scalability

To evaluate the scalability of the skip vector (marked as *SV* in the legend), we compare it against a lock-free skip list based on Fraser's algorithm [16] (*FSL* in the legend), taken from the Synchrobench suite [3]. Synchrobench has a few other skiplist implementations—notably, the no hotspot nonblocking skip list [6], the rotating skip list [15], and NUMASK [14], but these were consistently worse than Fraser for the workloads we evaluated. Unfortunately, the CUSL [22] is not publicly available. We produced a fair approximation (*USL* in the legend) by removing index-layer chunking from the SV implementation, to evaluate the impact of doing so.

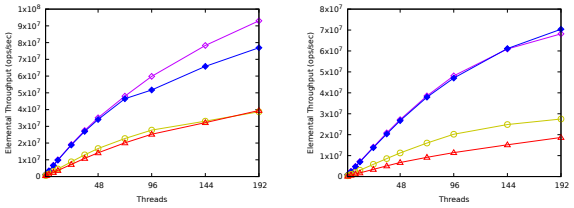
While the skip vector bears similarity to B+ trees, we were not able to find any correct, concurrent, high-performance open-source B+ trees to compare against. The closest options we found were the B+ tree by Braginsky and Petrank [25] and PALM [26]. In the former case, the implementation was by a third party, and was written in the Go language. Due to differences in language (i.e., garbage collection and the threading model), performance was not comparable. In the latter case, the implementation was also by a third party, and crashed at high thread counts.

We consider key ranges up to 2^{31}, at which point Fraser and the other competitors available in Synchrobench experienced out-of-memory errors; SV was able to complete for key ranges up to 2^{35}. The charts depict SV and USL variants which use hazard pointers to reclaim memory precisely (*HP* in the legend), and variants which leak memory (*Leak* in the legend). FSL does not reclaim memory, so only one variant is shown.

The skip vector has several tunable parameters; their impact is discussed in detail in Section V-B. We performed tuning experiments at every reported key range, and found an ideal tuning for each (marked *Tune* in the legend). For use cases where the number of elements is not known a priori, we also found general parameters that work well at most sizes: $\mathsf{layerCount} = 6$, $\mathsf{targetDataVectorSize} = 32$, $\mathsf{targetIndexVectorSize} = 32$, and $\mathsf{mergeThreshold} = 1.67 \times$

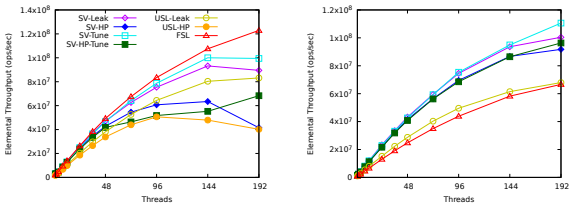


(a) 2^{20} keys (b) 2^{24} keys

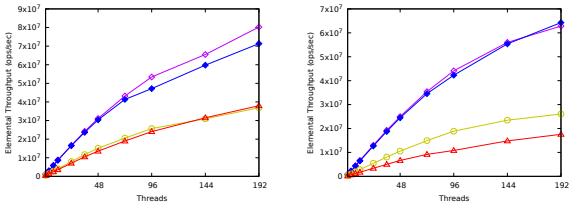


(c) 2^{28} keys (d) 2^{31} keys

Fig. 4: Throughput for an 80/10/10 lookup/insert/remove ratio.



(a) 2^{20} keys (b) 2^{24} keys



(c) 2^{28} keys (d) 2^{31} keys

Fig. 5: Throughput for a 0/50/50 workload.

targetSize.) In most cases, tuning did not significantly improve perfomance, so we omitted the tuned results. Additionally, USL–HP is omitted in charts where the worst skip vector outperforms USL–Leak, as it performed worse in all cases.

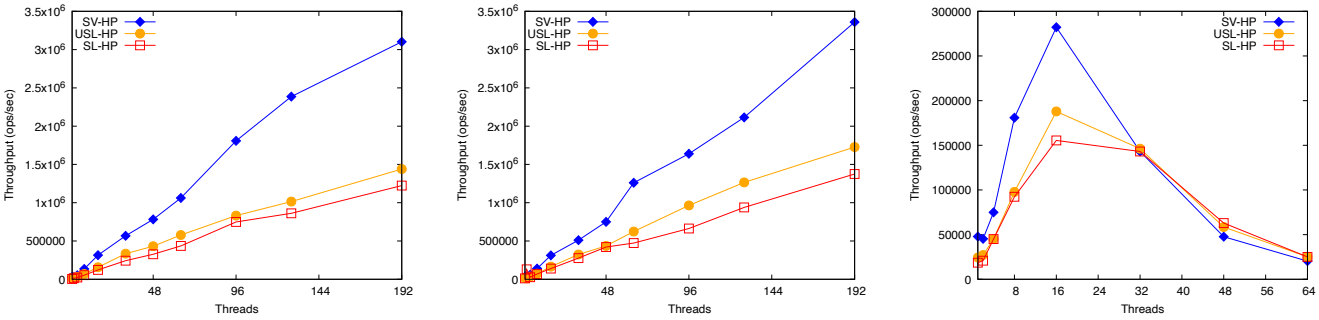
Figure 4 presents the performance on a microbenchmark performing a mix of 80% `Lookup`, 10% `Insert`, and 10% `Remove` (80/10/10), for key ranges 2^{20}, 2^{24}, 2^{28}, and 2^{31}. For figure 5, the ratio is 0% `Lookup`, 50% `Insert`, and 50% `Remove` (0/50/50). Operations are chosen randomly, and keys are drawn from a uniform distribution.

In an 80/10/10 workload mix with a key range of 2^{20} (Figure 4a), SV-Leak provides the best performance. SV-HP outperforms FSL up to 96 threads. It is possible to adjust parameters to outperform FSL (SV-HP-Tune, with parameters targetDataVectorSize $= 64$, mergeThreshold $= 1.0$, and layerCount $= 4$), but the larger point is that even with *blocking*, *memory reclamation* overheads, and default parameters, SV-HP is a solid performer. Lastly, we see that the skip vector always outperforms its USL equivalent. As the key range increases, the advantage of the skip vector only grows. At 2^{24} and 2^{28} keys (Figures 4b-4c), the skip vectors perform roughly twice as well as their competitors, even with precise memory reclamation and without special tuning. At 2^{31} (Figure 4d), the gap nears $3\times$.

For the 0/50/50 workload (Figure 5) the performance of the skip vector is worse overall than for 80/10/10. For 2^{24} keys and above, contention is low, and the higher locality of SV gives the best performance. At 2^{20} keys and more than 48 threads, the skip vector suffers, and FSL outperforms. A workload with a small key range and high modification rate is the worst case for the skip vector: While modifications to the topmost layer are rare, they are frequent enough to cause blocking and a higher rate of cache misses. Additionally, the granularity of contention is coarser for skip vectors than skip lists, where each element is independently synchronized. Furthermore, since the sequence locks are invisible to the operating system scheduler, we occasionally have a thread context switch while holding the top layer lock. These events, though uncommon, immediately disrupt performance. In contrast, nonblocking techniques have no such pathology. Note that even in this worst-case workload, chunking in the index layer is advantageous: the skip vector always outperforms the USL.

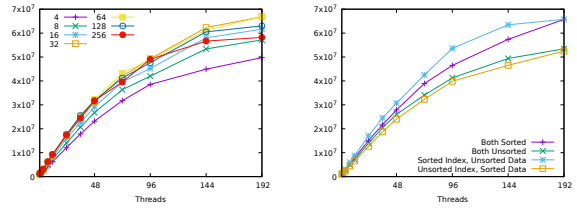
Figures 4 and 5 also show the cost of memory reclamation. While considerable at 2^{20} keys, it lessens significantly for higher key ranges, typically under 20%. At 2^{31}, the overhead of memory reclamation is negligible. This is a significant finding, since others have observed that linked data structures can suffer up to a $10\times$ slowdown with hazard pointers [27]. Second, we see that memory safety, in the form of hazard pointers, does not impact scalability. Thus we can conclude that the skip vector's design reduces the cost of memory safety, making it more viable than concurrent skiplists for languages with manual memory reclamation.

To evaluate skip vector perfomance under more realisitc conditions, we integrated SV-HP into DBx1000 [28], a single node OLTP in-memory database system from the YCSB benchmark suite. We also integrated as competitors two variants of SV-HP: the first removes index-layer chunking to simulate the unrolled skiplist (USL-HP) and the second does not chunck at all to simulate the traditional skiplist (SL-HP). The integrated data structures are used as indexes to accelerate access to keys. Figure 6 shows performance for a single table with 24M rows. Each thread runs 100K transactions, each of which accesses 16 rows. 90% of accesses are read-only, and keys are generated randomly using a Zipfian distribution with θ = 0.1, 0.6, and 0.9. When the distribution is not highly skewed (θ = 0.1 and θ = 0.6), chunking in both index and data layers causes almost $2\times$ higher throughput versus both USL-HP and SL-HP. In a skewed workload (θ = 0.9), a similar trend is observed for low thread count (less than 32 threads), then performance notably degrades, and all competitors. Our interpretation for this degradation is that the high contention on a small subset of keys changes the bottleneck to be the



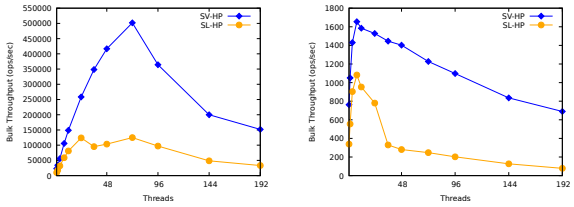
(a) θ = 0.1 (b) θ = 0.6 (c) θ = 0.9

Fig. 6: YCSB DBx1000 throughput.



(a) Index layer vector size (b) Vector configuration

Fig. 7: Configuration sensitivity for an 80/10/10 workload.



(a) 2^{12}-key range length (b) 2^{17}-key range length

Fig. 8: Throughput of all-range workloads with 2^{20} keys.

concurrency control layer of DBx1000.

B. Sensitivity to Configuration Parameters

In a skip list, the user must set the number of layers in advance, and must guess a high enough value to preserve asymptotic guarantees. The skip vector is less sensitive to incorrect guesses, since a "high" guess will not add many layers to the index. However, the skip vector can also be tuned by specifying values for targetIndexVectorSize, targetDataVectorSize, and mergeThreshold, as well as by choosing between sorted and unsorted vectors in the index and data layers.

We consider the impact of these parameters in Figure 7. The figure considers an 80/10/10 workload mix on a skip vector with hazard pointers, and a key range of 2^{28}. Figure 7a varies targetIndexVectorSize, while adjusting layerCount to the minimum value needed to maintain asymptotic guarantees and holding all other parameters constant. The worst configuration is about 25% slower than the best, with performance tapering off for values farther from the best configurations (32 and 64). The graph for targetDataVectorSize is similar but omitted for space. When these two parameters are too small, the skip vector cannot effectively exploit locality, and begins to behave like an ordinary skip list. When these two parameters are too big, vector operations become too expensive, and contention becomes a bigger problem, as modifications keep more elements locked for longer.

Figure 7b examines the use of sorted and unsorted vectors. The best performer uses sorted index vectors and unsorted data vectors. For sorted vectors, lookup is cheap (binary search is $O(log(T))$), but insertion and removal are expensive ($O(T)$). Thus, sorted vectors tend to be beneficial in the index layers, where lookups are common and modifications rare. While unsorted vectors cost $O(T)$ for all three common operations, insertion and removal have $O(1)$ write complexity, making them profitable in the data layer.

Most concurrent skip lists in research literature do not provide a lightweight mechanism for performing robust range operations. Some support atomic snapshots [29], and others support read-only range queries [30]. Since the skip vector is lock-based, it is trivial to employ two-phase locking to provide range queries. Figure 8 measures the throughput for mutating range queries. It compares the default skip vector against a tuned version that does not chunk at all (*SL*). This comparison minimizes differences in memory reclamation, and both implementations are serializable. The experiments are conducted for a key range of 2^{20}, and we consider operations over a small (2^{12}) or large (2^{17}) range. In both cases, the skip vector offers substantially higher throughput while the workload has parallelism to exploit. In the second chart there is little parallelism, since operations access $\frac{1}{8}$ of the total key range. For the shorter key range, the skip vector is able to scale to 72 threads before contention becomes an issue. Our previous work [8] discusses more advanced techniques for range operations, and is applicable to the skip vector.

VI. Conclusion

In this paper, we presented the skip vector, a concurrent map inspired by skip lists. In comparison to skip lists, the skip vector employs coarser synchronization metadata to achieve better spatial locality. This tradeoff is motivated by the realization

that conflicts are rare at deep layers, and writes are uncommon at shallow layers of skip lists. We showed that these locality-improving approaches are beneficial in both the index and data layers. The simplicity of the skip vector belies its effectiveness: its flexible design and high locality outperform the state of the art in skip lists, often by a large margin. It does so while using less memory, correctly managing and reclaiming its memory, and supporting linearizable range queries.

As future work, we plan to explore the interplay between the skip vector and emerging storage-class memory technologies (e.g., [31]–[33]). The dense packing of data in the skip vector is a good fit for these memories, which have higher latency than DRAM [34].

We also plan to investigate the use of the skip vector as a database index: Its flexible design can be tailored to the sorts of range queries needed by modern data processing systems, and its predictability and low latency make it an appealing choice for high-performance systems.

REFERENCES

[1] B. Stroustrup, "C++ Style (Keynote Address)," in *Proceedings of the 2012 GoingNative Conference*, 2012.

[2] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Communications of the ACM*, vol. 33, pp. 668–676, Jun. 1990.

[3] V. Gramoli, "More Than You Ever Wanted to Know about Synchronization," in *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*, San Francisco, CA, Feb. 2015.

[4] I. Lotan and N. Shavit, "Skiplist-Based Concurrent Priority Queues," in *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[5] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The SprayList: A Scalable Relaxed Priority Queue," in *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*, San Francisco, CA, Feb. 2015.

[6] T. Crain, V. Gramoli, and M. Raynal, "No Hot Spot Non-blocking Skip List," in *Proceedings of 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, Jul. 2013.

[7] 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.

[8] M. Rodriguez and M. Spear, "Optimizing Linearizable Bulk Operations on Data Structures," in *Proceedings of the 49th International Conference on Parallel Processing*, Edmonton, AB, Canada, Aug. 2020.

[9] M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, Jun. 2004.

[10] N. Cohen and E. Petrank, "Efficient Memory Management for Lock-Free Data Structures with Optimistic Access," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, Portland, OR, Jun. 2015.

[11] O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi, "Fast and Robust Memory Reclamation for Concurrent Data Structures," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, Asilomar State Beach, CA, Jul. 2016.

[12] H. Boehm, "Can Seqlocks Get Along with Programming Language Memory Models?" in *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, Beijing, China, Jun. 2012, pp. 12–20.

[13] M. Fomitchev and E. Ruppert, "Lock-Free Linked Lists and Skip Lists," in *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, Jul. 2004.

[14] H. Daly, A. Hassan, M. Spear, and R. Palmieri, "NUMASK: High Performance Scalable Skip List for NUMA," in *Proceedings of the 32nd International Symposium on Distributed Computing*, New Orleans, LA, Oct. 2018.

[15] I. Dick, A. Fekete, and V. Gramoli, "A Skip List for Multicore," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 4, 2017.

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

[17] A. Braginsky, A. Kogan, and E. Petrank, "Drop the Anchor: Lightweight Memory Management for Non-Blocking Data Structures," in *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, Montreal, Quebec, Canada, Jul. 2013.

[18] T. Brown, "Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way," in *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing*, Portland, OR, Jun. 2015.

[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] ISO/IEC JTC 1/SC 22/WG 21, "Technical Specification for C++ Extensions for Transactional Memory," May 2015. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf

[21] Y. Afek, H. Avni, and N. Shavit, "Towards Consistency Oblivious Programming," in *Proceedings of the 15th International Conference on Principles of Distributed Systems*, Toulouse, France, Dec. 2011.

[22] K. Platz, N. Mittal, and S. Venkatesan, "Concurrent Unrolled Skiplist," in *39th IEEE International Conference on Distributed Computing Systems*, Dallas, TX, Jul. 2019.

[23] N. Moscovici, N. Cohen, and E. Petrank, "A GPU-Friendly Skiplist Algorithm," in *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*, Portland, OR, Sep. 2017.

[24] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[25] A. Braginsky and E. Petrank, "A Lock-Free B+tree," in *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, Jun. 2012.

[26] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 795–806, 2011.

[27] T. Zhou, V. Luchangco, and M. Spear, "Hand-Over-Hand Transactions with Precise Memory Reclamation," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, Washington, DC, Jul. 2017.

[28] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 209–220, 2014.

[29] 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.

[30] 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.

[31] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, vol. 30, no. 1, 2010.

[32] Newsroom, Intel, "Intel and Micron produce breakthrough memory technology," 2018, https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[33] A. Tulapurkar, Y. Suzuki, A. Fukushima, H. Kubota, H. Maehara, K. Tsunekawa, D. Djayaprawira, N. Watanabe, and S. Yuasa, "Spin-torque diode effect in magnetic tunnel junctions," *Nature*, vol. 438, no. 7066, p. 339, 2005.

[34] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," 2019.