2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

Concurrent Unrolled Skiplist

Kenneth Platz
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75080, USA
kplatz@utdallas.edu

Neeraj Mittal
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75080, USA
neerajm@utdallas.edu

S. Venkatesan
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75080, USA
venky@utdallas.edu

***Abstract*—Skiplist is an important data structure used for storing and managing ordered data. It provides logarithmic time complexity (in list size) for lookup, insert and remove operations with high probability without the need for complex balancing actions. Several algorithms have been proposed for concurrent maintenance of a skiplist using both blocking and non-blocking synchronization techniques.**

In this work, we propose a new algorithm for maintaining a concurrent skiplist that uses two techniques to boost performance. The first technique, referred to as *unrolling*, involves storing multiple key-value pairs in the same node. The second technique involves using an advanced locking primitive, based on *group mutual exclusion*, to allow certain operations to work on the same node concurrently. In our experiments, our concurrent skiplist consistently outperformed existing concurrent skiplists by as much as 90% in some cases.

***Keywords*-concurrent data structure, skiplist, unrolling, group mutual exclusion**

I. Introduction

With the growing prevalence of multi-core, multi-processor systems, concurrent data structures are becoming increasingly important. In such a data structure, multiple processes may need to operate on the data structure at the same time. Contention between different processes must be managed in such a way that all operations complete correctly and leave the data structure in a valid state.

Skiplist is one of the fundamental data structures for organizing *ordered* data (a key-value pair) that supports lookup, insert and remove operations. It provides logarithmic time complexity in list size for the three operations with high probability without the need for complex balancing actions used by self-balancing search trees. Some applications of skiplists include Redis, skipdb and leveldb. In the last two decades, many algorithms have been developed for maintaining a concurrent skiplist using both blocking and nonblocking synchronization techniques [2], [7], [11], [20].

In [11], Fraser described three different concurrent algorithms for a skiplist based on (i) software transactional memory (STM), (ii) multi-word compare-and-swap (MWCAS) instruction, and (iii) single-word compare-and-swap (CAS) instruction. The last two are lock-free algorithms.

In [20], Lev, *et al.* presented a lock-based concurrent skiplist using lazy synchronization approach [16]. Intuitively, lazy synchronization approach involves the following high-level steps: (i) find the appropriate window (a pair of consecutive nodes) in each of the constituent linked list of the skiplist, (ii) lock the nodes in the relevant windows, (iii) ascertain that all the relevant windows are still valid, (iv) update the linked lists as in sequential counterpart, and (v) unlock all the nodes.

In [15], [16], Herlihy, *et al.* presented a lock-free concurrent skiplist partly based on the lock-free concurrent skiplist proposed by Fraser in [11]. It uses the lock-free linked list proposed by Michael in [21] as a building block.

In [7], Crain, *et al.* proposed a *No Hot Spot* lock-free skiplist in which adding or deleting an element from the skiplist is *decoupled* from the actions performed to maintain the structure of the skiplist. An insert operation simply splices a node into the linked list at the lowest level and a dedicated background process is responsible for restoring the topology of the skiplist by splicing the node into the other linked lists it is part of in a bottom-up manner. A remove operation simply marks a node for deletion and a dedicated background process is responsible for physically snipping out node from all the linked lists it is part of in a top-down manner.

In [2], Avni, *et al.* proposed a *LeapList*, which is a lock-based skiplist specifically designed to support fast *linearizable* range queries. Each node in a Leaplist holds up to K immutable key-value pairs. The algorithm utilizes software transactional memory for synchronization. Compared to other concurrent skiplists, it has somewhat faster lookup operations, slower update operations and much faster range query operations (which are also linearizable).

Our contributions: In this work, we present a new concurrent algorithm for maintaining a skiplist in a multicore system that uses two techniques to improve performance. The first technique, referred to as *unrolling*, involves storing multiple key-value pairs in the same node. This technique improves spatial locality and reduces the number of pointers that have to be followed in order to locate a given element. The second technique involves using an advanced locking primitive, based on *group mutual exclusion*, to allow certain operations to work on the same node concurrently. This technique reduces the contention window and increases concurrency. In our experiments, our concurrent skiplist, utilizing the ideas of unrolling as well as group mutual exclusion, consistently outperformed existing concurrent skiplists by as much as 90% in some cases.

This work was supported, in part, by the National Science Foundation (NSF) under grants numbered CNS-1115733 and CNS-1619197.

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00157

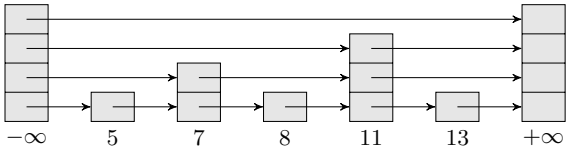


Fig. 1. Layout of a skiplist

Roadmap: The rest of the text is organized as follows. Section II describes our system model and some preliminary concepts. Section III describes our concurrent algorithm for maintaining a skiplist. In section IV, we experimentally evaluate our concurrent skiplist and compare its performance with that of existing skiplists. Finally, in section V, we present our conclusions and outline directions for future research.

II. System Model and Preliminaries

A. System Model

We consider an asynchronous shared-memory system consisting of n processes labeled $p_1, p_2, \ldots, p_n$. Each process also has its own private variables. A system execution is modeled as a sequence of process steps. In each step, a process either performs some local computation affecting only its private variables or executes one of the available instructions (read, write or synchronization primitive) on a shared variable. Processes take steps asynchronously. This means that in any execution, between two successive steps of a process, there can be an unbounded but finite number of steps performed by other processes. Every process is assumed to be *live* meaning that, if it has not terminated, then it will eventually execute its next step.

1) Synchronization Primitives: We assume the availability of *locks* to achieve mutual exclusion. At most one process can hold a lock at any time. Further, if one or processes requests a lock that is free, then some process is granted the lock eventually.

In addition to locks, we assume the availability of three read-modify-write (RMW) instructions, namely *fetch-and-increment (FAI)*, *fetch-and-decrement (FAD)* and *compare-and-swap (CAS)*.

A fetch-and-increment (respectively, fetch-and-decrement) instruction takes a shared variable x as input, returns the current value of x as output, and, at the same time, increments (respectively, decrements) the value of x by one.

A compare-and-swap instruction takes three arguments—a shared variable x and two values old and new—as input; it compares the value of x to old and, only if they are the same, modifies the value of x to new. It returns true if the value of x was modified and false otherwise.

B. Skiplist

A skiplist is a dictionary abstract data type that stores a set of key-value pairs and provides three operations, namely *lookup*, *insert* and *remove*. A *lookup* operation accepts a *key* as an argument; it returns either a valid value indicating success or a sentinel value, denoted by ϵ_v, indicating failure. An *insert* operation accepts a *key* and a *value* as arguments; it returns either true if the operation successfully inserted the key-value pair or false if the key was already present in the list. A *remove* operation accepts a *key* as argument; it returns either true if the operation successfully removed the key from the list and false if the key was not present in the list.

We refer to the key of an operation as its *target key*.

Internally, a skiplist consists of a *hierarchy* of linked lists. Each linked list stores elements in an order sorted by their keys and the list size decreases by a constant factor at every level as we move up the hierarchy. The linked list at the lowest level contains every element in the dictionary. The linked list at the next higher level contains a subset of the elements contained in the linked list immediately below it (the former "skips over" some elements in the latter). This property is referred to as *inclusion invariant*, which ensures that every element present at level $i + 1$ is also be present at levels $0 \ldots i$.

Given a linked list and a key, let $pred$ and $curr$ denote the two consecutive nodes in the list such that the given key is strictly greater than the key stored in $pred$ but smaller than or equal to that stored in $curr$. We refer to the pair $(pred, curr)$ as the *window* of the list with respect to the given key. Note that if the given key is present in the list then it must match the key stored in $curr$.

As an example, traversing the skiplist in fig. 1 with target key of 8 would return the windows $(-\infty, +\infty)$, $(-\infty, 11)$, $(7, 11)$ and $(7, 8)$ (starting from the topmost level).

To locate a given element, a process successively traverses all the linked lists of the skiplist one-by-one, starting with the list at the highest level and moving its way downward. At each level, the process finds the window for the list at that level *using* the window for the list previously found at the immediately higher level. We refer to the rightmost node in the lowest level window as the *target node*.

Inserting an element in a skiplist involves selecting the topmost level for the node containing the target key using certain random distribution and then *splicing* the node into each of the linked lists in a bottom-up manner.

Removing an element from a skiplist involves *snipping out* the node containing the target key from each of the linked list in which it is present in a top-down manner. But, before a node is snipped out of any list, it is *marked* (*i.e.*, logically deleted).

C. Unrolling a List

Unrolling a list involves storing multiple key-value pairs in a single node [24]. This has several advantages. First, it reduces the number of pointers that need to be followed to locate a given key. This, in turn, reduces the cache coherence traffic. Second, in most cases, an update (insert or remove) operation does not cause any structural changes to the list and can be completed by locking a single node only (in a concurrent environment).

D. Group Mutual Exclusion

Storing multiple key-value pairs in a single node increases the likelihood of conflicts among update operations since two

update operations would now conflict if their target nodes are same even if their target keys are different. To minimize conflict, we observe that multiple update operations can be allowed to act on the same node concurrently provided they are of the same type (either all are insert or all are remove). However, to implement this technique, we need a different type of lock that allows a request to specify a type (in our case insert or remove) so that requests of the same type can hold a lock simultaneously. Note that a read-write lock is not sufficient because it allows only one type of request, namely read, to share a lock. The problem of building this type of lock has been well-studied in the literature in the form of *group mutual exclusion* although, to our knowledge, no implementation of this lock currently exists in any operating system.

Many algorithms have been proposed for solving the GME problem [3], [8], [13], [14], [18], [19]. Unfortunately, most of them require $\Omega(n)$ space per lock making them unsuitable for our case [3], [8], [13], [18]. This is because each node in skiplist is protected using a separate instance of a lock. A skiplist may contain a large number of nodes (a million or more). Using a GME algorithm that has $\Omega(n)$ space complexity per instance would increase the space usage of an unrolled skiplist significantly. A few existing GME algorithm have incremental space complexity of $O(1)$ per lock (in the asymptotic case) [14], [19]. In our concurrent algorithm, we use two different implementations of GME algorithm. The first algorithm is analogous to the well-known test-test-and-set algorithm [16] for a traditional lock and the second algorithm is Keane and Moir's GME algorithm adapted for multiple locks [19].

a) Test-Test-And-Set (TTAS) Based GME Algorithm: Our simple algorithm uses a single word to keep track of (a) type of session in progress and (b) the number of processes in the session. To join a session, a process repeatedly reads the contents of the word until it finds that either there is no session in progress or the session currently in progress is the one it wants to join. In both cases, it attempts to update the contents of the word using a CAS instruction. Specifically, in the first case, it tries to set the session type to that of is request and the session count to one. In the second case, it tries to increment session count by one. If the CAS instruction succeeds, then the process has successfully joined the session and can start executing its critical section. Otherwise, it repeats the above steps. To leave a session, a process repeatedly attempts to update the contents of the word using a CAS instruction until it succeeds. Specifically, it tries to decrement the session count by one. And, if it is the last process to leave the session, then it also tries to set the session type to a sentinel value at the same time.

b) Keane and Moir's GME Algorithm: In Keane and Moir's GME algorithm [19], an exclusive lock is used to protect access the code for joining and leaving a session. When trying to join a session, if a process finds that the current session is not compatible with that of its request, then it inserts itself into a queue. It then spins on a flag until a new session compatible with its own request is established and the flag is set to an appropriate value (by the process that established the session). Although the algorithm as presented in [19] assumes a single lock, it can be easily adapted for multiple locks, while achieving $O(1)$ (asymptotic) space complexity per lock, by sharing the queue among all locks.

In addition to having low space complexity per lock, we selected this algorithm because it has low step complexity especially in the absence of contention.

E. Correctness Conditions

We use *linearizability* as the safety property and *deadlock-freedom* as the liveness property [16]. Roughly speaking, linearizability states that, in every execution of the system, each operation should appear to take effect at some instant of time between when it started and when it ended. Further, deadlock-freedom states that, in every infinite execution of the system, whenever one or more processes have a pending operation then some operation should eventually complete thereafter.

III. Our Concurrent Algorithm

A. Unrolling a Skiplist

In our earlier work [23], we developed a lock-based concurrent algorithm for maintaining an unrolled linked list using lazy synchronization technique [16]. In this work, we present a lock-based concurrent algorithm for maintaining an *unrolled skiplist* again using lazy synchronization technique. Although the two algorithms are similar at high level, they differ in details. This is because a skiplist is a more complicated data structure as it consists of multiple (but related) linked lists. As a result, our concurrent algorithm for an unrolled linked list cannot be directly applied to a skiplist. Some of the differences are as follows. First, in our concurrent algorithm for an unrolled linked list, the lock associated with a node protects only the next pointer of the node. However, in the concurrent algorithm for an unrolled skiplist, the lock associated with a node protects all the contents of the node. Second, due to the way a skiplist is organized, we lock nodes in skiplist in tail-to-head order as opposed to head-to-tail order in linked list. Third, merge and rebalance procedures have been modified to use predecessor node instead of successor node.

Each node stores up to K key-value pairs. Each node is associated with a special *immutable* key, referred to as *anchor* key. In any linked list, nodes are arranged in the order of their anchor keys. However, no order is maintained among the regular keys stored within the same node. We maintain the following three invariants. First, every regular key stored in a node is greater than or equal to the anchor key associated with the node. Second, any regular key stored in a node is strictly less than the anchor key associated with its successor node. Third, each node in the skiplist stores a minimum number of valid key-value pairs that is within some constant factor of K.

For ease of exposition, we assume that a key-value pair can be stored in a single word and hence can be read and written *atomically*. If not, this can be easily achieved by associating

a monotonically increasing timestamp with each slot in the node [16].

As in traditional skiplist, we use two sentinel nodes, namely *head* and *tail*; both nodes are permanently part of the skiplist and are contained in linked lists at all levels. An empty skiplist is initialized with three nodes, the two sentinel nodes head and tail and a regular node with anchor keys $\perp_0$, $\perp_1$ and $\top_0$, respectively, where $\perp_0 < \perp_1 < \top_0$. Further any regular key lies between $\perp_1$ and $\top_0$. Moreover, we represent an empty slot using (ϵ_k, ϵ_v) as the key-value pair.

The traversal in an unrolled skiplist, which we refer to as *scan*, works the same way as in traditional skiplist except we use anchor keys for comparison with the target key.

As in traditional skiplist, a scan of an unrolled skiplist returns an array of windows—one for each level in the skiplist. Recall that a window is a pair of consecutive nodes in a linked list such that if the key is indeed present in the linked list, then it is guaranteed to be contained in the second node of the pair. We refer to the second node of the window for the linked list at the lowest level as the *target node*.

Lookup operation: A lookup operation inspects the target node to check whether any slot in the node contains the target key. If it finds such a slot, it returns the value associated with the slot. Otherwise, it returns the sentinel value ϵ_v.

Insert operation: An insert operation locks the target node, ascertains that the node is not marked and inspects the node to check whether any slot in the node contains the target key. While inspecting the node, it also keeps track of any empty slot in the node. If it fails to find a slot containing the target key, then there are two possible cases. If the operation was able to find an empty slot, then it stores the desired key-value pair in that slot. Otherwise, the node is full and needs to be *split*. A split procedure involves creating a pair of nodes and distributing the key-value pairs stored in the target node roughly equally among the two nodes, while maintaining the invariants with respect to anchor keys described earlier. The desired key-value pair is then stored in one of the two nodes. Finally, the target node is marked and *replaced* with the pair of new nodes (explained later).

Remove operation: A remove operation locks the target node, ascertains that the node is not marked and inspects the node to check whether any slot in the node contains the target key. If it finds a slot containing the target key, it deletes the key-value pair from the slot and updates the node count. If, after deleting the pair, the node has too many empty slots, then structural changes are required to the skiplist to ensure that the minimum occupancy invariant is not violated. This is accomplished by either one of the following: moving all the valid key-value pairs of the target node to its predecessor (if it has enough number of empty slots) or redistributing the keys of the target node and its predecessor among a pair of two new nodes. We refer to the former as *merge* and the latter as *rebalance*. In the merge procedure, the target node is marked and snipped out of the skiplist. In the rebalance procedure, the target node and its predecessor are marked and replaced with the pair of new nodes (explained later).

Algorithm 1: Data structures

```
1  struct Node {
       // number of valid data elements (key-value pairs)
          in the node
2      int count;
       // lock to protect the contents of the node
3      Lock lock;
       // the highest level that the node is part of
4      int level;
5      Data data[K]; // array of key-value pairs
       // whether or not the node is marked for deletion
6      bool marked;
       // next pointer for each level node is part of
7      NodePtr next[TOPLEVEL];
8  };
9  struct State {
10     Key key; // target key of the operation
11     Value value; // value associated with the key if any
       // arrays of pointers used in traversal
12     NodePtr ppreds[TOPLEVEL];
13     NodePtr preds[TOPLEVEL];
14     NodePtr currs[TOPLEVEL];
15     NodePtr succs[TOPLEVEL];
16 };
   // Local variables used by a process
17 StatePtr state := create a new state record;
18 Key key;
19 Value value;
20 NodePtr ppred, pred, curr, succ, node1, node2;
21 NodePtr[] tmp;
22 int slot, result, level, maxlevel, minlevel;
```

Replacement procedure: In split and rebalance procedures, we need to replace either a single node (split) or a pair of nodes (rebalance) with another pair of nodes. This needs to be done carefully because node(s) being replaced may contain keys besides the target key. To ensure that our algorithm generates only linearizable executions, we need to ensure that keys other than target keys are *continuously* present in the dictionary even during the replacement procedure.

To that end, we proceed as follows. First, we lock all the relevant nodes—whose next pointers will undergo a change—in tail-to-head order. Second, we ascertain that none of the locked nodes is marked as well as none of the next pointers that will be updated has changed since they were last observed during the traversal. Third, we snip out the node(s) being replaced from all the linked lists they are part of except for the list at the lowest level in top-down manner. Fourth, we snip out the node(s) being replaced *as well as* splice the nodes replacing then into the lowest level linked list in a *single* step by switching the appropriate next pointer. Fifth, we splice the replacing nodes into the linked lists they need to be part of in bottom-up manner. Finally, we release all the locks.

B. Formal Description

A pseudocode of our algorithm is given in Algorithms 1-7. For ease of exposition, we assume that locks are reentrant.

Algorithm 1 shows the data structures used by our algorithm: a *list node* and a *state record*. A list node is a shared object. A state record is local to a process and is used to store information relevant to an operation including the windows located during the most recent traversal of the skiplist.

Algorithm 2 shows the two functions used to traverse a skiplist. The first function `Scan` locates the window at each level of the skiplist in a top-down manner. The second function `Seek` searches the target node (the rightmost node in the lowest-level window) for a given key. If it finds the key, it

Algorithm 2: Functions for traversing the skiplist

```
   // locates the window at each level
23 Scan(StatePtr state)
24 begin
25     ppred := null;
26     pred := head ;
27     for ℓ ← TopLevel down to 0 do
28         curr := pred.next[ℓ];
29         succ := curr.next[ℓ];
30         while (succ.anchor ≤ state.key) do
31             ppred := pred;
32             pred := curr;
33             curr := succ;
34             succ := succ.next[ℓ];
35         state.ppreds[ℓ] := ppred;
36         state.preds[ℓ] := pred;
37         state.currs[ℓ] := curr;
38         state.succs[ℓ] := succ;
       // fix windows at higher levels
39     for ℓ ← TopLevel down to curr.level +1 do
40         state.preds[ℓ] := state.currs[ℓ];
41         state.currs[ℓ] := state.succs[ℓ];
       // fix predecessor's predecessor pointers at higher
          levels
42     for ℓ ← TopLevel down to pred.level +1 do
43         state.ppreds[ℓ] := state.preds[ℓ];
   // examines the window at the lowest level to locate
      the target key if present; otherwise, it returns the
      location of an empty slot if found
44 (Value, int) Seek(StatePtr state)
45 begin
46     slot := K;
47     value := ϵ_v;
48     curr := state.currs[0];
49     for i ← 0 to K − 1 do
50         (key, value) := curr.data[i];
51         if (state.key = key) then break;
52         else if ((key = ϵ_k) and (slot = K)) then slot := i ;
53     return (value, slot);
```

Algorithm 3: Lookup, insert and remove operations

```
54 Value Lookup(Key key)
55 begin
56     state.key := key;
57     Scan(state);
58     (value, __) := Seek(state);
59     return value;
60 bool Insert(Key key, Value value)
61 begin
62     state.key := key;
63     state.value := value;
64     while true do
65         Scan(state);
66         Acquire(currs[0].lock);
67         result := Add(state);
68         Release(currs[0].lock);
69         if result = FAILURE then continue;
70         else return result = ADDED ? true : false;
71 bool Remove(Key key)
72 begin
73     state.key := key;
74     while true do
75         Scan(state);
76         Acquire(currs[0].lock);
77         result := Delete(state);
78         Release(currs[0].lock);
79         if result = FAILURE then continue;
80         else return result = DELETED ? true : false;
```

returns the value associated with the key. Otherwise, it returns the location of an empty slot in the data array, if one exists.

Algorithm 3 shows the high-level pseudocode for the three skiplist operations `Lookup`, `Insert` and `Remove`. In all three operations, a process first invokes `Scan` function to locate the window at each level of the skiplist. In `Lookup` operation, the process then invokes `Seek` function to locate the data element with the given key in the target node. In `Insert` and `Remove` operations, on the other hand, the process then locks the target node and invokes either `Add` function (if `Insert` operation) or `Delete` function (if `Remove` operation).

Algorithm 4: Functions for locking and unlocking nodes at multiple levels

```
81 bool LockAndValidateUntil(NodePtr[] from, to, int level)
82 begin
83     for ℓ ← 0 to level do
84         Acquire(from[ℓ].lock);
85         result := not(from[ℓ].marked);
86         result := result and (from[ℓ].next[ℓ] = to[ℓ]);
87         if not(result) then
88             UnlockUntil(from, ℓ);
89             return false;
90     return true;
91 UnlockUntil(NodePtr[] from, int level)
92 begin
93     for ℓ ← level down to 0 do Release(from[ℓ].lock) ;
```

Algorithm 5: Helper functions for insert and remove operations

```
94 int Add(StatePtr state)
95 begin
96     (value, slot) := Seek(state);
97     if (value ≠ ϵ_v) then return PRESENT;
98     curr := state.currs[0];
99     if (curr.marked) then return FAILURE;
100    if (slot < K) then
101        curr.data[slot] := (state.key, state.value);
102        curr.count := curr.count +1;
103        return ADDED;
       // need to make structural changes to the skiplist
104    if Split(state) then return ADDED;
105    else return FAILURE;
106 int Delete(Key key)
107 begin
108    (value, slot) := Seek(state);
109    if (value = ϵ_v) then return ABSENT;
110    curr := state.currs[0];
111    if (curr.marked) then return FAILURE;
112    if (curr.count > LowerBound) or (curr.anchor = ⊥_1) then
113        curr.data[slot] := (ϵ_k,ϵ_v);
114        curr.count := curr.count −1;
115        return DELETED;
       // need to make structural changes to the skiplist
       // lock the predecessor node
116    if not(LockAndValidateUntil(state.preds, state.currs, 0)) then
         return FAILURE;
117    pred := state.preds[0];
118    if (pred.count + curr.count ≤ UpperBound) then
119        result := Merge(state)
120    else result := Rebalance(state) ;
121    UnlockUntil(state.preds, 0);
122    if (result then return DELETED;
123    else return FAILURE;
```

Algorithm 4 shows the pseudocode for locking and unlocking an array of nodes (corresponding to predecessors of the *leftmost* node being deleted from the skiplist) until a given level. In the first function `LockAndValidateUntil`, a process locks the nodes in a bottom-up manner. As it locks a node, it ascertains that the node is not marked and its next pointer has not changed. In the second function `UnlockUntil`, a process unlocks previously locked nodes in a top-down manner.

Algorithm 5 shows the pseudocode of helper functions `Add` and `Delete`. In `Add` function, a process first checks if the

Algorithm 6: Split procedure

```
124 bool Split(StatePtr state)
125 begin
126     level := select a level number randomly;
127     curr := state.curr_s[0];
128     maxlevel := max{level, curr.level};
129     minlevel := min{level, curr.level};
        result := LockAndValidateUntil(state.preds,
                                       state.curr_s, maxlevel);
        if (result) then
            create two new nodes, say node1 and node2;
            distribute data elements in curr evenly among node1 and node2
              such that all keys in node1 are smaller than those in node2;
            node1.anchor := curr.anchor;
            node2.anchor := smallest key value in node2;
            copy (state.key, state.value) to node1 or node2 as
              appropriate;
            initialize node1.count and node2.count appropriately;
            node1.level := minlevel;
            node2.level := maxlevel;
            Acquire(node1.lock);
            Acquire(node2.lock);
            // set up next pointers of node1 and node2
130         for ℓ ← 0 to node1.level do node1.next[ℓ] := node2;
131         for ℓ ← 0 to node2.level do
132             if (ℓ ≤ curr.level) then node2.next[ℓ] := curr.next[ℓ] ;
133             else node2.next[ℓ] := preds[ℓ].next[ℓ] ;
            // snip out curr from the skiplist
134         curr.marked := true;
135         for ℓ ← curr.level down to 1 do
136             preds[ℓ].next[ℓ] := curr.next[ℓ];
            // splice node1 and node2 into the skiplist
137         preds[0].next[0] := node1;
138         for ℓ ← 1 to maxlevel do
139             if (ℓ > node1.level) then
140                 preds[ℓ].next[ℓ] := node2;
141             else preds[ℓ].next[ℓ] := node1;
142         Release(node1.lock);
143         Release(node2.lock);
144         UnlockUntil(state.preds, maxlevel);
        return result;
```

target node already contains a data element with the given key. If not and the target node is not marked, it either stores the given key-value pair in an empty slot, if one exists, or invokes `Split` function, if not. In `Remove` function, a process first checks if the target node contains a data element with the given key. If yes and the target node is not marked, it either deletes the data element, if the node contains enough data elements or has special anchor key, or invokes `Merge` or `Rebalance` functions, otherwise.

Algorithm 6 shows that pseudocode for `Split` function used by `Insert` operation. In `Split` function, a process first locks all the nodes whose next pointer(s) will undergo a change and performs validation. If the validation succeeds, it then creates two new nodes and distributes all the data elements stored in the target node evenly among the two new nodes along with the key-value pair to be added to the skiplist. It then replaces the target node with the two new nodes using the replacement procedure described earlier.

Algorithm 7 shows that pseudocode for `Merge` and `Rebalance` functions used by `Remove` operation. In both functions, a process first locks all the nodes whose next pointer(s) will undergo a change and performs validation. In `Merge` function, if the validation succeeds, then the process copies all data elements from the target node to its predecessor and removes the target node. In `Rebalance` function, if the validation succeeds, then the process creates two new nodes and distributes all the data elements stored in the target node

Algorithm 7: Merge and rebalance procedures

```
145 bool Merge(StatePtr state)
146 begin
147     pred := state.preds[0];
148     curr := state.curr_s[0];
149     level := curr.level;
        result := LockAndValidateUntil(state.preds,
                                       state.curr_s, maxlevel);
        if (result) then
            copy all data elements from curr to pred;
            // snip curr out from the skiplist
150         curr.marked := true;
151         for ℓ ← level down to 0 do
152             preds[ℓ].next[ℓ] := curr.next[ℓ];
153         UnlockUntil(state.preds, level)
        return result;

154 bool Rebalance(StatePtr state)
155 begin
156     pred := state.preds[0];
157     curr := state.curr_s[0];
158     maxlevel := max{pred.level, curr.level };
        // create an array that contains the relevant next
           pointers of state.ppreds
159     for ℓ ← 0 to pred.level do tmp[ℓ] := pred;
160     for ℓ ← pred.level +1 to curr.level do tmp[ℓ] := curr;
        result := LockAndValidateUntil(state.ppreds,
                                       tmp, maxlevel);
        if (result) then
            create two new nodes, say node1 and node2;
            distribute data elements in pred and curr, except for the data
              element with key state.key, evenly among node1 and node2;
            node1.anchor := pred.anchor;
            node2.anchor := smallest key in node2;
            node1.level := pred.level;
            node2.level := curr.level;
            initialize node1.count and node2.count appropriately;
            Acquire(node1.lock);
            Acquire(node2.lock);
            // set up next pointers of node1 and node2
161         for ℓ ← 0 to node1.level do
162             if (ℓ ≤ node2.level) then
163                 node1.next[ℓ] := node2;
164             else node1.next[ℓ] := pred.next[ℓ] ;
165         for ℓ ← 0 to node2.level do
166             node2.next[ℓ] := curr.next[ℓ];
167         pred.marked := true;
168         curr.marked := true;
            // snip pred and curr out of the skiplist
169         for ℓ ← maxlevel down to 1 do
170             if (ℓ > curr.level) then
171                 ppreds[ℓ].next[ℓ] := pred.next[ℓ];
172             else ppreds[ℓ].next[ℓ] := curr.next[ℓ] ;
            // splice node1 and node2 into the skiplist
173         ppreds[0].next[0] := node1;
174         for ℓ ← 1 to maxlevel do
175             if (ℓ ≤ node1.level) then
176                 ppreds[ℓ].next[ℓ] := node1;
177             else ppreds[ℓ].next[ℓ] := node2;
178         Release(node1.lock);
179         Release(node2.lock);
180         UnlockUntil(state.ppreds, maxlevel);
        return result;
```

and its predecessor, except for the data element whose key is being deleted from the skiplist, evenly among the two new nodes. It then replaces the target node and its predecessor with the two new nodes using the replacement procedure described earlier.

C. Proof of Correctness

Due to space constraints, we only provide a proof sketch here.

1) All executions are linearizable: For convenience, we assume that skiplist only stores keys and not key-value pairs. We first give some definitions.

Algorithm 8: Insert and remove operations using a GME lock

```
181 bool Insert(Key key, Value value)
182 begin
183     state.key := key;
184     state.value := value;
185     while (true) do
186         Scan(state);
187         curr := currs[0];
188         Acquire(curr.lock, INSERTMODE); // in shared mode
189         (value, slot) := Seek(state);
190         if (value ≠ ϵv) then
191             Release(curr.lock);
192             return false;
193         if (curr.marked) then
194             Release(curr.lock);
195             continue;
196         for i ← slot to K-1 do
197             if CAS(curr.data[slot], (ϵk,ϵv), (key,value)) then
198                 FAI(curr.count);
199                 Release(curr.lock);
200                 return true;
201             else if (curr.data[slot] = (key, __) then return false;
202         Release(curr.lock);
203         Acquire(curr.lock); // in exclusive mode
204         result := Add(state); // call the helper function
205         Release(curr.lock);
206         if (result = FAILURE) then continue;
207         else return result = ADDED ? true : false;
208 bool Remove(Key key)
209 begin
210     state.key := key;
211     while (true) do
212         Scan(state);
213         curr := currs[0];
214         Acquire(curr.lock, REMOVEMODE); // in shared mode
215         (value, slot) := Seek(state);
216         if (value = ϵv) then
217             Release(curr.lock);
218             return false;
219         if (curr.marked) then
220             Release(curr.lock);
221             continue;
222         if (curr.count > LOWERBOUND) or (curr.anchor = ⊥1) then
223             if CAS(curr.data[slot], (key,value), (ϵk,ϵv)) then
224                 FAD(curr.count);
225                 Release(curr.lock);
226                 return true;
227             else return false;
228         Release(curr.lock);
229         Acquire(curr.lock); // in exclusive mode
230         result := Remove(state); // call the helper function
231         if (result = FAILURE) then continue;
232         else return result = DELETED ? true : false;
```

Definition 1 (weakly sorted unrolled linked list)**.** *We say that an unrolled linked list is* weakly sorted *if (a) the value of every key stored in the node is greater than or equal to that of node's anchor key, and (b) the value of every key stored in the node is strictly less than that of its successor's anchor key (if the successor exists).*

Definition 2 (weakly sorted unrolled skiplist)**.** *We say that a hierarchy of linked lists forms a* weakly sorted unrolled skiplist *if (a) the linked list at every level is a weakly sorted unrolled linked list, and (b) at every level of the hierarchy except for the lowest level, the linked list at that level is a sublist of the linked list at the level immediately below.*

Definition 3 (active/passive node)**.** *We say that the node is* active *if it is reachable from the head node; otherwise, we say that the node is* passive.

Let A be a node and let $level(A)$ denote the highest-level linked list to which A belongs. It can be proved using induction that our algorithm maintains the following invariants with respect to A.

(I1) For each $\ell \in [0, level(A)]$, if A is the tail node, then A has no successor at level ℓ.

(I2) For each $\ell \in [0, level(A)]$, if A is a non-tail node, then A has a non-null successor at level ℓ.

(I3) Every key stored in node A is greater than or equal to $anchor(A)$.

(I4) The anchor key of node A's successor at level 0 is monotonically non-decreasing over time.

(I5) For each $\ell \in [0, level(A)]$, every key stored in node A is strictly smaller that the anchor key of the A's successor at level ℓ.

At any level, we refer to the chain of nodes from the head node to the tail node as *active chain*. Using the definition of active chain and invariants defined above, we can infer that any active chain is a weakly sorted unrolled linked list. Clearly, our algorithm maintains the inclusion invariant. Thus, the collection of active chains, one at each level, forms a weekly sorted unrolled skiplist.

Note that any write step is only performed on an active node. We now argue that the outcome of any traversal of the skiplist is "correct". A single traversal consists of one instance of scan procedure followed by one instance of seek procedure. The following lemmas can be proved using induction.

Lemma 1. *If the target key of a traversal is continuously present in the skiplist during the entire duration of the traversal, then the traversal is guaranteed to find the key.*

Lemma 2. *If the target key of a traversal is continuously absent from the skiplist during the entire duration of the traversal, then the traversal is guaranteed to not find the key.*

Our algorithm supports three types of operations: lookup, insert and remove. We categorize the lookup operations into two types: a *lookup-hit* operation finds its target key and a *lookup-miss* operation does not. For ease of exposition, we treat insert and remove operations that return false as lookup-hit and lookup-miss operations, respectively. To complete the proof, we now specify the *linearization point* of each operation.

Insert operation: If the insert operation performs a split procedure, then the linearization point of the operation is the point when it spliced the pair of new nodes into the lowest level linked list. Otherwise, it is the point when it stored its target key in an empty slot in the target node.

Remove operation: The linearization point of the operation is the point when it erased the slot in the target node containing its target key.

Lookup-hit operation: In this case, we can infer that the key was not continuously absent from the skiplist during the traversal performed by the lookup operation. If the key was present in the skiplist when the traversal started, then the linearization point of the lookup operation is the point when the traversal started. Otherwise, at least one insert operation

with the same target key must have its linearization point during the traversal. Thus, the linearization point of the lookup operation is the point immediately after the linearization point of any such insert operation.

Lookup-miss operation: In this case, we can infer that the key was not continuously present in the skiplist during the traversal performed by the lookup operation.If the key was absent in the skiplist when the traversal started, then the linearization point of the lookup operation is the point when the traversal started. Otherwise, at least one remove operation with the same target key must have its linearization point during the traversal. Thus, the linearization point of the lookup operation is the point immediately after the linearization point of any such remove operation.

It can be proved that the sequential history obtained by ordering operations based on their linearization points is legal.

2) All executions are deadlock-free: The proof is by contradiction. Assume that some infinite execution of the system is not deadlock-free. It implies that the system eventually reaches a state after which no operation completes even though some process has a pending operation in that state. This, in turn, implies that the system eventually reaches a state whereafter every process with a pending operation is either (a) blocked forever waiting on a lock, or (b) repeatedly traversing the skiplist, acquiring the requisite locks and then releasing the locks due to failed validation. We refer to this state as *quiescent* state. Note that all states reachable from a quiescent state are also quiescent.

Let B and R denote the set of processes belonging to each category. We first argue that $R \neq \emptyset$. Let $S \subseteq B$ denote the subset of processes in B that are waiting on the lock associated with the node with the *smallest* anchor key. If R is empty, then, clearly, some process in S is eventually granted the lock because, by our assumption, every lock is deadlock-free—a contradiction. Thus R contains at least one process, say p. Consider the traversal of p that starts in a quiescent state. Note that, once the system has reached a quiescent state, thereafter (a) the skiplist cannot undergo any change, and (b) no node in the skiplist is marked. Otherwise, it would imply that some update (insert or remove) operation is current modifying the skiplist and would eventually complete—a contradiction. Clearly, this in turn implies that the validation procedure performed by p is guaranteed to succeed. Therefore p eventually completes its update operation—a contradiction.

D. Enhancing Intra-Node Concurrency using Group Mutual Exclusion

A node now stores multiple key-value pairs. As a result, two update operations conflict if their target nodes are same even if their target keys are different and the likelihood of conflict grows as node size increases. To mitigate this problem, we allow multiple update operations to work in a node concurrently as long as they are of the same type (all insert or all remove). This requires a different type of lock based on group mutual exclusion, which allows a process to specify a *type* with its request and two critical sections can overlap if they are of the same type.

Note that even if only operations are of the same type are allowed to act on a node concurrently, we still need to manage conflicts among them. To that end, we make the following modifications to our concurrent algorithm. First, a slot is now updated using a CAS instruction and not a simple write instruction. Second, whenever a new key-value pair is stored in an empty slot, the node count is incremented using a FAI instruction. Likewise, when an existing key-value pair is deleted from a slot, the node count is decremented using a FAD instruction. Third, if a process finds that it needs to make structural changes to the skiplist (execute split, merge or rebalance procedure), then it releases its lock on the target node, which was acquired in shared mode, and reacquires it in exclusive mode. Besides the above changes, each update operation needs to be slightly modified. We only describe the parts that need to be changed.

An insert operation inspects the slots in the target node in a certain order, say from left to right, to locate an empty slot. On finding an empty slot, it attempts to store the desired key-value pair in the empty slot using a CAS instruction. If it fails, then it means that another key-value pair has been stored in that slot. If the key now present in the slot matches the operation's target key, the operation terminates and returns false. Otherwise, it resumes its inspection from that slot onward.

A remove operation inspects the slots in the target node to determine if any slot contains the target key. If it finds such a slot, it attempts to erase the key-value pair from the slot using a CAS instruction. If it fails, then it means that another operation has erased the pair from the slot; the former operation terminates and returns false.

Algorithm 8 shows the pseudocode of `Insert` and `Remove` operations using GME-based locks.

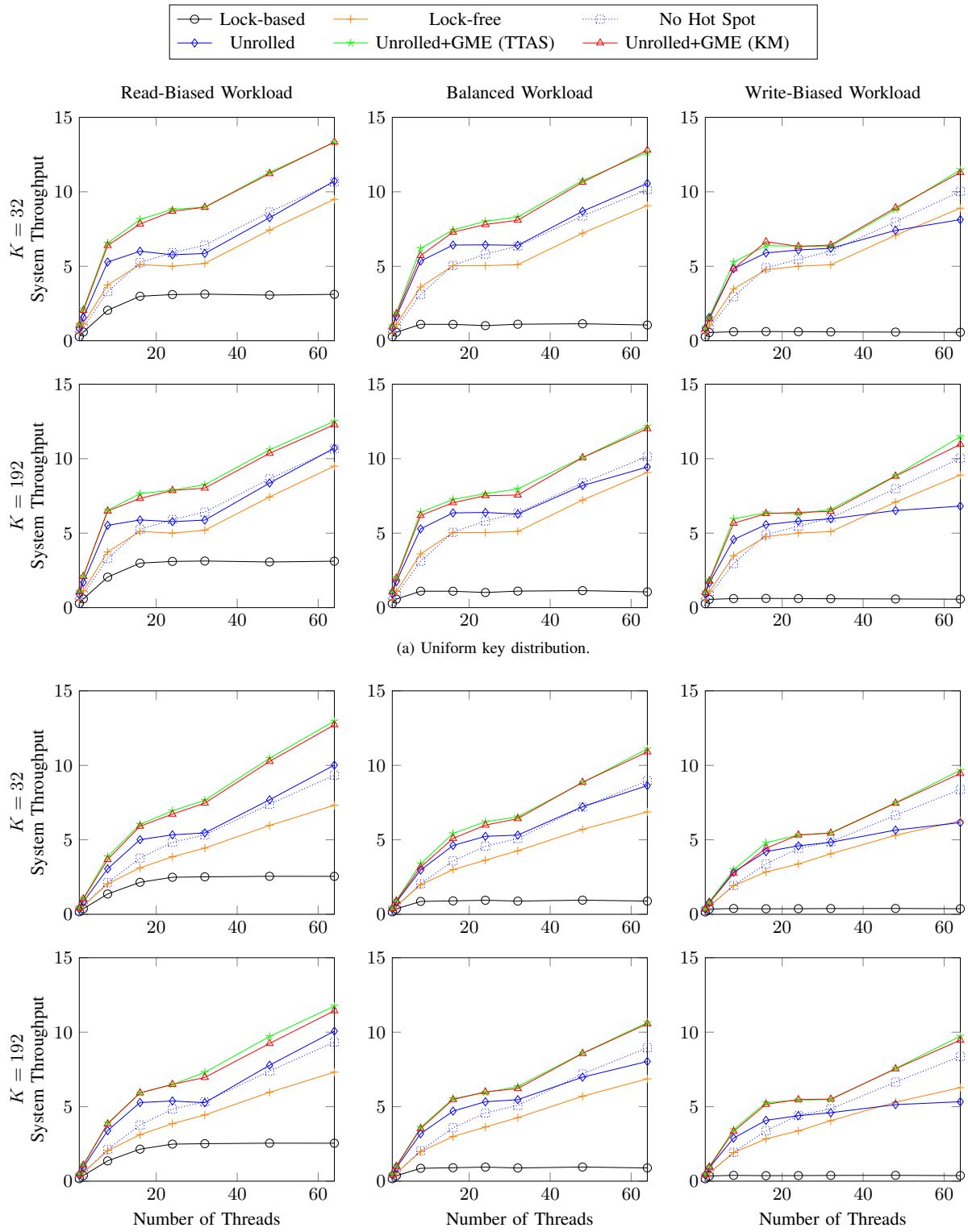
IV. Experimental Evaluation

We now describe the results of evaluating different concurrent skiplists using simulated workloads.

A. Concurrent Skiplist Implementations

We compare the performance of the following implementations of concurrent skiplists: (i) a lock-based skiplist by Herlihy, *et al.* [20], denoted by LBSkipList, (ii) a lock-free skiplist proposed by Fraser [11], denoted by LFSkipList, (iii) a lock-free "No Hot Spot" skiplist proposed by Crain, *et al.* [7], denoted by NHSSkipList, (iv) unrolled skiplist (described in this work), denoted by UnrolledSkipList, (v) unrolled skiplist using naïve TTAS-based GME algorithm, denoted by TTASBasedSkipList, and (vi) unrolled skiplist using Keane and Moir's queue-based GME algorithm [19], denoted by KMBasedSkipList.

The first three implementations were obtained from Synchrobench [12]. All implementations are written in C/C++. The three implementations from Synchrobench use keys only and not key-value pairs. We thus modified our implementations to use keys only.



(a) Uniform key distribution.

(b) Zipfian key distribution with $\alpha = 0.5$.

Fig. 2. Variation in system throughput (reported in number of operations completed per microsecond) with the number of threads.

We do not reclaim memory of garbage objects in our experiments. Recently, two general approaches for memory

reclamation have been proposed based on the signaling mechanism of a Unix/Linux system [1], [5]. In fact, when we linked our implementations with ThreadScan library [1], we saw a relatively small performance drop of at most 10%. However, we were unable to link some of the Synchrobench implementations with ThreadScan library, and thus we disabled memory reclamation for all implementations. We believe that our experimental results should still stand even when memory reclamation is used.

B. Experimental Setup

We conducted our experiments on a dual socket Intel Xeon E5-2698 v3 processor consisting of 16 2.3 GHz cores per socket with hyperthreading enabled (yielding 64 logical cores in total) and 512GB RAM. We used g++ compiler with optimization flags set to -O3 -march=native -funroll-loops. We used GSL (GNU Scientific Library) to generate random numbers. To avoid memory allocation in a multi-threaded environment from becoming a bottleneck, we used Facebook's *jemalloc* [9] as the dynamic memory allocator since it provides superior performance to the C/C+ default allocator in a multi-threaded environment.

To comparatively evaluate different implementations, we considered the following parameters:

1) **Maximum List Size:** This depends on the size of the key space. We considered key space size of 10^7 (ten million) keys.
2) **Maximum Node Size:** This depends on the number of key-value pairs a node can store. We considered two different node sizes of 32 and 192.
3) **Relative Distribution of Operations:** We considered three different workload distributions: (a) *read-biased:* 90% lookup, 5% insert and 5% remove, (b) *balanced:* 50% search, 25% insert and 25% remove, and (c) *write-biased:* 0% lookup, 50% insert and 50% remove.
4) **Maximum Degree of Concurrency (or Contention):** This depends on number of threads that can concurrently operate on the skiplist. Since our system has 64 cores, we varied the number of threads from 1 to 64 in suitable increments.

In each run of the experiment, we ran an implementation for ten seconds and averaged the results over ten runs. The beginning of each run consisted of a one second "warm-up" phase whose numbers were excluded from the calculations to minimize the effect of initial caching on the computed statistics. Also, to capture only the steady state behavior, we *pre-populated* the skiplist to 50% of its maximum size, prior to starting a simulation run.

a) Key Distribution: Typically, when evaluating a dictionary type data structure, keys are assumed to be uniformly distributed, *i.e.*, all keys in the key space are assumed to have the same probability of occurrence [10], [17], [22]. But, many real-world workloads are better modeled using skewed distributions in which some keys are more popular than others [4], [6]. Zipfian (or simply Zipf) distribution is a type of power-law distribution that simulates this behavior [4]. It is characterized by a parameter α that usually lies between 0.5 and 1 [4]. In our experiments, we used both uniform and Zipfian (with $\alpha = 0.5$) distributions to evaluate the performance of different skiplists.

b) Evaluation Metrics: We compared the performance of different implementations with respect to *system throughput*, which is given by the number of operations completed per unit time.

C. Results

Figure 2 shows the results for uniform and Zipfian key distributions, respectively. The top and bottom rows of plots in each figure show the results for the two different node sizes of $K = 32$ and $K = 192$, respectively.

As the graphs show, the two GME-based skiplists, TTAS-BASEDSKIPLIST and KMBASEDSKIPLIST, *consistently* outperformed the other four skiplists, UNROLLEDSKIPLIST, NHSSKIPLIST, LFSKIPLISTand LBSKIPLIST, for all key space sizes, node sizes and workloads used in our experiments. The skiplist using TTAS-based GME locks had marginally better performance than the one using Keane and Moir's algorithm based GME locks. The *next best* performer was either UNROLLEDSKIPLIST (under low or medium contention) or NHSSKIPLIST (under high contention especially when the workload was write-biased and the number of threads was large). Between the two lock-free skiplists, NHSSKIPLIST outperformed LFSKIPLIST under almost all scenarios except when the number of threads was relatively small. The lock-based traditional skiplist, LBSKIPLIST, had the worst-performance under all scenarios. Further, for all different variants of unrolled skiplist, the variant with $K = 32$ had slightly better performance than the corresponding variant with $K = 192$ in most cases.

A detailed analysis of experimental data showed that the relative gap in the throughput between unrolled skiplists and the next best performing skiplist increased as the number of threads decreased. Figure 3 shows the relative throughput for selected skiplist implementations when the workload was read-biased; other workloads displayed a similar trend albeit the gap became narrower as the workload became more write-heavy. This can be explained as follows. When the number of threads is small, unrolling significantly reduces the traversal time of an operation due to fewer pointer dereferences and improved spatial data locality. However, as the number of threads increases, the contention between operations also grows because the likelihood that two operations have the same target node becomes higher. However, using GME-based locks helps mitigate this problem by allowing more operations to work on the same target node concurrently in most cases (unless structural changes to the list are required). As the graphs show, for write-based workload, even though at 64 threads lock-free skiplists had much better performance than unrolled skiplist using traditional (ME-based) locks, they were still outperformed by the two unrolled skiplists using GME-based locks.

Number of Threads	LFSkipList	NHSSkipList	UnrolledSkipList ($K = 32$)	TTASBasedSkipList ($K = 32$)
2	1.00	0.79	1.40	1.91
4	1.00	0.81	1.40	1.88
8	1.00	0.88	1.41	1.75
16	0.98	1.00	1.14	1.55
32	0.81	1.00	0.91	1.40
64	0.89	1.00	1.00	1.25

(a) Uniform key distribution.

Number of Threads	LFSkipList	NHSSkipList	UnrolledSkipList ($K = 32$)	TTASBasedSkipList ($K = 32$)
2	1.00	0.95	1.45	1.84
4	1.00	0.98	1.45	1.85
8	0.97	1.00	1.44	1.82
16	0.83	1.00	1.33	1.60
32	0.83	1.00	1.03	1.44
64	0.78	1.00	1.07	1.39

(b) Zipfian key distribution with $\alpha = 0.5$.

Fig. 3. Relative throughput of different concurrent skiplists for read-biased workload *with respect to* the best performer between LFSkipList and NHSSkipList.

V. Conclusions and Future Work

In this work, we have proposed a new concurrent algorithm for skiplist that uses two techniques to achieve better performance—unrolling and group mutual exclusion. In our experiments, our concurrent skiplist consistently outperformed existing concurrent skiplists by as much as 90% in some cases.

As future work, we plan to develop a group mutual exclusion algorithm that provides stronger progress guarantees, has lower space complexity per lock as well as lower step complexity in the absence of contention. We also plan to develop a lock-free version of our unrolled skiplist.

References

[1] Dan Alistarh, W. M. Leiserson, A. Matveev, and N. Shavit. ThreadScan: Automatic and Scalable Memory Reclamation. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 123–132, June 2015.

[2] H. Avni, N. Shavit, and A. Suissa. Leaplist: Lessons Learned in Designing TM-Supported Range Queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 299–308, 2013.

[3] V. Bhatt and C. C. Huang. Group Mutual Exclusion in $O(\log n)$ RMR. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 45–54, July 2010.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the 18th IEEE Conference on Computer Communications (INFOCOM)*, pages 126–134, March 1999.

[5] T. A. Brown. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 261–270, July 2015.

[6] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-Law Distributions in Empirical Data. *SIAM Review*, pages 661–703, 2009.

[7] T. Crain, V. Gramoli, and M. Raynal. No Hot Spot Non-Blocking Skip List. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 196–205, 2013.

[8] R. Danek and V. Hadzilacos. Local-Spin Group Mutual Exclusion Algorithms. In *Proceedings of the 18th Symposium on Distributed Computing (DISC)*, pages 71–85, October 2004.

[9] J. DeLeong. Folly: The Facebook Open Source Library. https://www.facebook.com/notes/facebook-engineering/folly-the-facebook-open-source-library/10150864656793920/, 2012.

[10] D. Drachsler, M. Vechev, and E. Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 343–356, February 2014.

[11] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, February 2004.

[12] V. Gramoli. More than You Ever Wanted to Know about Synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, January 2015.

[13] V. Hadzilacos. A Note on Group Mutual Exclusion. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, August 2001.

[14] Y. He, K. Gopalakrishnan, and E. Gafni. Group Mutual Exclusion in Linear Time and Space. In *Proceedings of the 17th International Conference on Distributed Computing and Networking (ICDCN)*, January 2016.

[15] M. Herlihy, Y. Lev, and N. Shavit. A Lock-Free Concurrent Skiplist with Wait-Free Search. Unpublished Manuscript, Sun Microsystems Laboratories, 2007.

[16] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.

[17] S. V. Howley and J. Jones. A Non-Blocking Internal Binary Search Tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 161–171, June 2012.

[18] P. Jayanti, S. Petrovic, and K. Tan. Fair Group Mutual Exclusion. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 275–284, July 2003.

[19] P. Keane and M. Moir. A Simple Local-Spin Group Mutual Exclusion Algorithm. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 23–32, 1999.

[20] Y. Lev, M. Herlihy, V. Luchangco, and N. Shavit. A Simple Optimistic Skiplist Algorithm. In *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 124–138, Castiglioncello, Italy, June 2007.

[21] M. M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-based Sets. In *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 73–82, 2002.

[22] A. Natarajan and N. Mittal. Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, February 2014.

[23] K. Platz, N. Mittal, and S. Venkatesan. Practical Concurrent Unrolled Linked Lists Using Lazy Synchronization. In *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS)*, pages 388–403, 2014.

[24] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling Lists. In *Proceedings of the ACM Conference on LISP and Functional Programming (LFP)*, pages 185–195, New York, NY, USA, 1994. ACM.