A Skip List for Multicore

Ian Dick
University of Sydney

Alan Fekete
University of Sydney

Vincent Gramoli
University of Sydney

Abstract

In this paper, we introduce the Rotating skip list, the fastest concurrent skip list to date. Existing concurrent data structures experience limited scalability with the growing core count for two main reasons: threads contend while accessing the same shared data and they require off-chip communication to synchronize.

Our solution combines the rotation of a tree to maintain logarithmic complexity deterministically, a skip list structure to avoid tree root bottleneck, and no locks to limit cache line bouncing. This combination requires us to trade usual skip list towers for wheels, a novel algorithmic design that favors spatial locality and allows for a constant-time restructuring operation.

We evaluate the performance of our skip list on AMD Opteron and Intel Xeon multicores, show that its rotations guarantee its balance and compare its performance against 7 state-of-the-art skip lists and trees using 4 different synchronisation techniques. The Rotating skip list shows an unprecedented peak performance of 200 Mops/second.

1 Introduction

Modern chip multiprocessors offer an increasing amount of cores that share a memory large enough to store database indexes. The trend of increasing core counts rather than core frequency raises new research challenges to reach an unprecedented level of *throughput*, the number of operations or transactions executed per second. The crux of the problem is *contention* as multiple threads compete on the same data. The problem is more visible on the inherent bottlenecks of data structures. A thread updating the root to rebalance a tree will typically contend with all concurrent traversals.

A skip list [28] is an interesting alternative data structure as it also offers logarithmic complexity, keeps its elements sorted in successive towers, but does not require a global rebalancing that creates contention bottlenecks. In addition, each of its nodes points to a successor node and a lower node similarly to tree nodes pointing to a left child and a right child except that the lower nodes is immutable whereas the left child is mutable. This immutability reduces contention further and is the reason why the JDK logarithmic concurrent structures are skip lists, not trees. Recently, a Java skip lists [9] avoids contention hotspots by relying on a maintenance thread to raise, lower and clean-up the towers. This technique lets application threads simply insert an element at the bottom or delete an element by marking it as logically deleted. Yet, typical skip lists, except in rare cases [9], do not exploit multicore platforms as well as trees.

In this paper, we introduce the *Rotating skip list*, a new skip list that combines the rotation of trees and the uncontended nature of skip lists into a multicore-friendly data structure. The core algorithmic novelty is the use of *wheels* instead of the usual towers that are linked together to speedup traversals. With classic towers, in a new traversal there is at least one access to shared memory per level of the skip list. By contrast, our wheels maximize cache hits by occupying contiguous memory and offering spatial locality. Each cache line of our AMD and Intel multicores spans multiple wheel items so that traversing the rotating skip list often leads to access the lower level item directly from the cache. As we measure in Section 5.4, wheels divide the cache miss rate by 2.8× compared to the no hotspot skip list [9].

Moreover, wheels are easily adjustable, in particular, decreasing the height of all wheels can be achieved in a single atomic step. Existing skip lists require either a linear time to adjust the size of all the towers, simply because each tower height gets adjusted individually [13, 29, 9], or require synchronization to prevent concurrent threads from adjusting contiguous towers. By contrast, the size of our wheels is dynamically adjusted using a global ZERO marker that wraps around the wheel capacity using modulo arithmetic to indicate the lowest level node of all wheels. This marker allows to lower the level of all wheels in one atomic step: all traversals observing the effect of this step simply ignore the bottom level. Once no threads are traversing the deleted level anymore, a dedicated background thread progressively deallocates it. To decrease contention further, we raise wheels similarly to the way towers are raised in the no hotspot skip list [9]: update operations only update the bottommost level while a maintenance thread raises towers in the background.

We integrated the Rotating skip list implementation in the publicly available Synchrobench [16] to compare its performance against the most efficient skip lists we are aware of: Fraser's [15] and Crain et al.'s [9]. The former is an efficient implementation in C whose peak performance still outperforms recent alternatives [2, 10] whereas the latter is the recent no hotspots skip list written in Java that outperforms the JDK skip lists by $2\times$ on multicore. Besides outperforming Fraser's on read-only workloads, our algorithm is well-suited to handle multicore contention: it is faster than Crain et al.'s non-blocking no hotspot skip list and is $2.6\times$ faster than Fraser's implementation when 64 cores access a 2^{10}-element key-value store with 30% update.

In addition, we compared the Rotating skip list with other data structures and synchronisation techniques, including read-copy-update (RCU) and transactional memory (TM). Overall, we show that the resulting logarithmic data structure is more efficient on two multicore machines, from AMD and Intel, than six logarithmic data structures exploiting four different synchronization techniques [15, 18, 12, 8, 9, 1] and we outline that its performance are comparable, under uniform workloads, to the performance of a lock-free binary search tree [11] that cannot be rebalanced.

We also show that our Rotating skip list is correct and discuss its complexity. In particular, we observe empirically that its access time complexity remains logarithmic and we show that it is non-blocking. An interesting drawback of our design decisions is that its space complexity is $O(n \log n)$ as opposed to the traditional $O(n)$ space complexity. While this might be a drawback to leverage the memory, our Rotating skip list implements an in-memory key-value store with unprecedented peak performance of 200 Mops/sec. We conclude from this extensive evaluation that well-engineered skip lists can be better suited than trees for multicores, as opposed to recent observations [30, 22]. Note that we could not compare against this latter work [22] as the code was not disclosed.

In Section 2, we state the problem. In Section 3, we detail how the rotating skip list implements a key value store. In Section 4, we present the experimental environment, the machines, the benchmarks and the tested data structures. In Section 5, we thoroughly evaluate the performance of non-blocking skip lists. In Section 6, we compare the performance to balanced trees and various synchronization techniques. In Section 7, we discuss the correctness and progress of the skip list and in Section 8, we discuss the complexity of the skip list and experiment the effectiveness of the background thread to restructure it. In Section 9, we present the related work and Section 10 concludes.

2 Problem Statement

Our key challenge is to enhance the throughput one can obtain from a concurrent data structure with logarithmic time complexity, referred to as *logarithmic data structures*. We are not aware of any such structure that are cache-friendly and free from contention hotspots.

Tolerance to contention. To measure the impact of this problem, let us select the state-of-the-art skip list from Fraser [15] and the recent no hotspots skip list from Crain et al. [9]. On the one hand, we confirm that Crain's skip list [9] performs well on a relatively small data structure where contention is high but we also observed that its performance degrades on large datasets. On the other hand, Fraser's skip list [15], which is known as the skip list with the highest peak throughput, is appropriately tuned for x86 architectures so to scale to large datasets, however it does not avoid hotspots and its performance degrades under contention.

We tried to understand this phenomenon by comparing the performance of Crain's and Fraser's skip lists on the publicly available Synchrobench benchmark suite [16]. For the sake of integration with Synchrobench, we ported Crain's skip list algorithm from Java to C, and we took Fraser's original C implementation [14], to experiment on a 64-core AMD machine (cf. experimental settings in Section 4).

The performance of Crain's and Fraser's skip lists are depicted in Figure 1. As expected the C implementation of Crain's skip list outperforms Fraser's on relatively small (2^{10}) datasets as its operations attempt to CAS few memory locations at the bottom of the skip list and avoid hotspots at the upper levels. In addition, we observed that Fraser's skip list performs better than Crain's on large datasets (set size 2^{16}). Our conclusion is that Crain's poor locality of reference entails a large amount of cache misses (especially when the cache space is likely to be exhausted due to large datasets).

Leveraging caching. We measured the number (and proportion) of cache misses during the execution of Crain's skip list. We observe that the rate of cache misses is low (less than 1 in 1000 in all our measurements) but it grows substantially between a structure of 2^{10} elements and a structure of 2^{16} elements: with a read-only workload, the miss rate increased by a factor of 4.6. Although the update percentage seems to be positively correlated to the cache miss rate, the size of the data structure is a more important factor influencing cache-misses. As more data is stored in memory, the probability of cache misses increases (cf. Table 1).

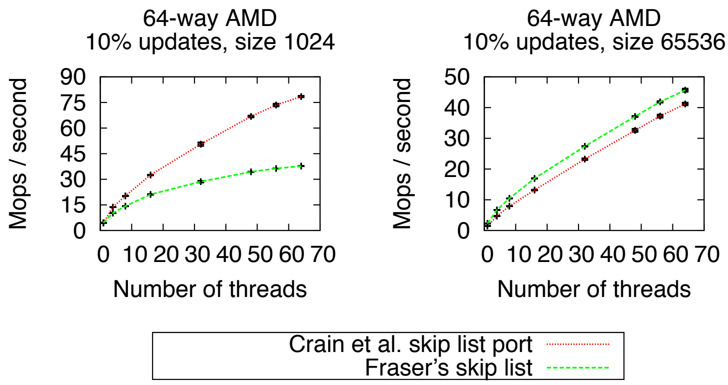


Figure 1: The C implementation of Crain's skip list does not perform well on large dataset while Fraser's skip list does not perform well under contention (update ratio is 10%)

update	0%	10%	30%
2^{10} size	(7,201K) 0.07‰	(7,469K) 0.12‰	(7,767K) 0.16‰
2^{16} size	(29,834K) 0.39‰	(35,457K) 0.83‰	(41,027K) 1.07‰
increase	4.6×	5.9×	5.7×

Table 1: Increase in cache miss rate as Crain's structure grows (absolute cache miss numbers in parentheses and the last row shows increase in the cache miss rate)

In Crain's skip list, the index elements of the list are represented using distinct memory structures to represent objects, which were connected to one another using rightwards and downwards pointers to represent the original object connected to one another by reference. This implementation makes the level of indirections, which do not leverage spatial locality, quite high.

Memory reclamation. As recently noted, memory reclamation is still a difficult problem to solve while providing progress guarantees as needed in non-blocking data structures [23]. Crain's skip list algorithm was proposed assuming the presence of a built-in stop-the-world garbage collector, as in Java. Unmanaged languages do not offer this feature and memory management implementations may impact drastically the performance, as it was already observed for simpler non-blocking data structures [26].

3 The Rotating Skip List

In this section, we present a key-value store implemented with our rotating skip list. The rotating skip list differs from traditional skip lists in that it is deterministic and uses wheels, its rotations differ from trees rotations in that they execute in constant time to lower the structure. As shown in Section 7, our resulting algorithm is linearizable and non-blocking (or lock-free).

3.1 Key-value store

Key-value stores offer the basis for indexes to speed up access to data sets. They support the following operations: (i) `put(k,v)` inserts the key-value pair $\langle k, v\rangle$ and returns `true` if k is absent, otherwise return $\perp$, (ii) `delete(k)` removes k and its associated value and returns `true` if k is present in the store, otherwise return `false`, (iii) `get(k)` returns the value associated with key k if k is present in the store, otherwise return `false`.

The concurrent implementation of the key-value store has to guarantee that the store behaves atomically or equivalently as if it was executing in a sequential environment. To this end, we require our key-value store implementation to be *linearizable* [20]. Informally, we require the aforementioned operations, when invoked by concurrent threads, to appear as if they had occurred instantaneously at some point between their invocation and response.

3.2 Structure

Memory locality is achieved by using a rotating array sub-structure, called the *wheel*, detailed in Section 3.4.

Algorithm 1 The rotating skip list - state and get function

```
1: State of the algorithm:
2:   sl, the skip list
3:   ZERO, a global counter, initially 0
4:   node is a record with fields:
5:     k ∈ ℕ, the key
6:     v, the value of the node or
7:       ⊥, if node is logically deleted                 ▷ logical deletion mark
8:       node, if node is physically deleted             ▷ physical deletion mark
9:     level, the number of levels in the node's wheels
10:    succs, the array of level pointers to next wheels ▷ the node's wheel
11:    next, the pointer to the immediate next node
12:    prev, the pointer to the immediate preceding node
13:    marker ∈ {true, false}, whether node is
14:      a marker node used to avoid lost
15:      insertion during physical removal

16: get(k):
17:   zero ← ZERO                                        ▷ logical zero index
18:   i ← sl.head.level − 1                              ▷ start at the topmost level
19:   item ← sl.head                                     ▷ start from the skip list head
20:   while true do                                      ▷ find entry to node level
21:     next ← item.succs[(zero + i) % max_levels]       ▷ follow level in wheels
22:     if next = ⊥ ∨ next.k > k then                    ▷ if rows ends here or key is larger
23:       next ← item                                    ▷ go backward once
24:       if i = zero then                               ▷ if bottom is reached
25:         node ← item                                  ▷ position reached
26:         break                                        ▷ done traversing index levels
27:       else i ← i − 1                                 ▷ move down a level
28:     item ← next                                      ▷ move to the right
29:   while true do                                      ▷ find the correct node
30:     while node = (val ← node.v) do                   ▷ physically deleted?
31:       node ← node.prev                               ▷ backtrack to the first non phys. deleted node
32:     next ← node.next                                 ▷ next becomes immediate next node
33:     if next ≠ ⊥ then                                 ▷ if next is non logically deleted
34:       next_v ← next.v                                ▷ check its value
35:       if next = next_v then                          ▷ if next is physically deleted
36:         help_remove(node, next)                      ▷ help remove deleted nodes...
37:         continue                                     ▷ ...then retry
38:     if next = ⊥ ∨ next.k > k then                    ▷ still at the right position?
39:       if k = node.k ∧ val ≠ ⊥ then return val        ▷ non deleted key found
40:       else return ⊥                                  ▷ logically deleted or not found
41:     node ← next                                      ▷ continue the traversal
```

The structure is a skip list, denoted *sl*, specified with a set of nodes, including two sentinel nodes. The *head* node is used as an entry point for all accesses to the data structure, it stores a dummy key that is the lowest of all possible keys. A *tail* node is used to indicate the end of the data structure, its dummy key is strictly larger than any other possible keys. On Figure 2 the head node is represented by the extreme left wheel whereas the tail is depicted by the extreme right wheel. As in other skip lists, node values are ordered in increasing key order from left to right. The global counter ZERO indicates the index of the first level in the wheels, it is set to 0 intially.

The *node* structure contains multiple fields depicted in Algorithm 1 at lines 1–15. It first contains a key-value pair denoted by $\langle k, v\rangle$. Two special values $v = \perp$ and $v = node$ indicate that the node is logically deleted and physically deleted, respectively. Note that the same special value $v = nodes$ is already used in the ConcurrentSkipListSet of the JDK by Doug Lea to avoid marking the node by setting the the low-order bit of a memory work, however, in C/C++ and on x86-64 architectures one could easily use the low-order bit technique instead. A node is first logically deleted before being physically removed and two logically deleted nodes cannot share the same key k. The *level* represents the level of this node's wheel, similar to the level of the corresponding tower in a traditional skip list, it indicates that the node keeps track of successors indexed from 0 to $level - 1$. The *succs* is the node's wheel, it stores the successor pointer at each level of the node. *next* (resp. *prev*) is a direct pointer to the next (resp. previous) node that contains the smallest key larger than k (resp. the highest key lower than k). Hence the skip list nodes are all linked through a doubly linked list as depicted at the bottom of Figure 3(a). This doubly linked list allows to backtrack among preceding nodes if the traversal ends up at a deleted node. Finally, the *marker* is a special mark used only during physical removal.

Algorithm 2 The rotating skip list - put and delete functions

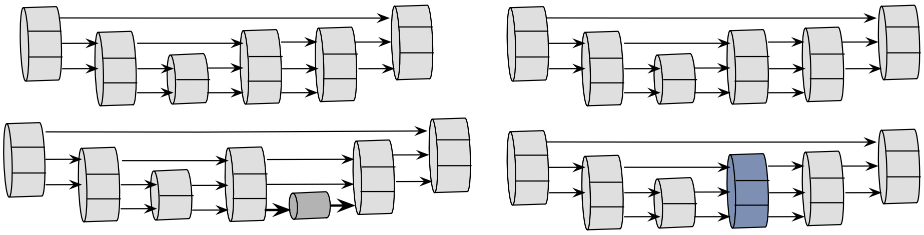
```
put(k, v):
  Find entry to bottom level as lines 17–28
  while true do                                          ▷ find the correct node
    while node = (val ← node.v) do                       ▷ physically removed?
      node ← node.prev                                   ▷ backtrack
    next ← node.next
    if next ≠ ⊥ then
      next_v ← next.v
      if next = next_v then
        help_remove(node, next)                          ▷ help remove marked nodes...
        continue                                         ▷ ...then retry
    if next = ⊥ ∨ next.k > k then                        ▷ position found
      if k = node.k then                                 ▷ key k is already present
        if val = ⊥ then                                  ▷ if logically deleted...
          if CAS(&node.v, val, v) then return true       ▷ ...logically insert
        else return false
      else                                               ▷ key is not already in the list
        new ← new_node(k, v, node, next)                 ▷ create new node
        if CAS(&node.next, next, new) then               ▷ physically insert
          if next ≠ ⊥ then next.prev ← new               ▷ link backward pointer
          return true                                    ▷ insertion succeeded
        else delete_node(new)                            ▷ contention, retry insertion
    node ← next                                          ▷ continue the traversal

delete(k):
  Find entry to bottom level as lines 17–28
  while true do                                          ▷ find the correct node
    while node = (val ← node.v) do                       ▷ physically removed?
      node ← node.prev                                   ▷ backtrack
    next ← node.next
    if next ≠ ⊥ then
      next_v ← next.v
      if next = next_v then
        help_remove(node, next)                          ▷ help remove marked nodes
        continue                                         ▷ ...then retry
    if next = ⊥ ∨ next.k > k then                        ▷ position found
      if k ≠ node.k then return false                    ▷ key k is not here
      else
        if val ≠ ⊥ then
          while true do                                  ▷ until node is logically deleted
            val ← node.v
            if val = ⊥ ∨ node = val then                 ▷ already deleted or removed
              return false
            else if CAS(&node.v, val, ⊥) then            ▷ logical delete
              if too-many-deleted() then                 ▷ heuristic
                remove(node.prev, node)                  ▷ removal
              return true                                ▷ logically deleted or physically removed
        else return false
    node ← next                                          ▷ continue the traversal
```

3.3 Traversal

Each update operation (put, delete) avoids contention hotspots by localizing the modification to the least contended part of the data structure (Alg. 1 and 2). All adjustments to the upper levels are sequentially executed by a dedicated maintenance thread, described in Section 3.5, hence allowing a deterministic adjustment of the levels.

Any traversal, whether it is for updating or simply searching the structure, is executed from the top of the *head* node traversing wheels from left to right and levels from top to bottom. Each access looks for the position of some key k in the skip list by starting from the top level of the head down to the bottom level. The pseudocode of the get function is described in Algorithm 1, lines 16–41. The get function starts by traversing the structure from the skip list head, namely $sl.head$, till the bottom level, as indicated lines 17–28. It records the value of ZERO at the beginning of the traversal into a local $zero$ variable, sets its starting point to the $set.head$ before iterating over each level i from the top level of the skip list, which is also the top level of the head $set.head.level - 1$, to $zero$.

Once the get has reached the bottom level, $node$ is actually set to the node with the largest key $k' < k$. If this $node$ is physically deleted, the traversal backtracks among deleted nodes at lines 30 and 31 and invokes help_remove to notify the background thread of the nodes to be removed (line 36). Note that the traversal can



(a) A put consists of adding a new wheel at the bottom level (before it gets raised)

(b) A delete consists of marking the node with the corresponding key as logically deleted (before getting physically remove)

Figure 2: Adding a new key-value pair consists of inserting a node at the bottom-most level while removing a pair consists of marking its node

thus reach a node that is not linked by any *succs* pointer but only one *next* pointer (e.g., the 3rd node of the skip list of Figure 3(c)). Then it updates *next* to the immediate next node (*node*.*next*). Once the right position at the bottom level indicated by $next.k > k$ (line 38) is reached, the targeted key k is checked (line 39). If it is non logically deleted, the associated value *val* is returned, otherwise $\perp$ is returned to indicate that no key k was present in the key-value store.

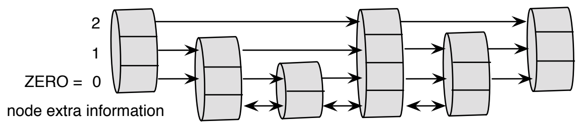
The put function (Algorithm 2, lines 42–64) is similar in that it first traverses the structure from top to bottom (line 43), backtracks from right to left (line 46) and help_remove deleted nodes (line 51). The put may find that the node with the key it looks for is logically deleted (line 55), in which case it simply needs to logically insert it by setting its value to the appropriate one using a CAS (line 56). if the node is found as non logically deleted, then put is unsuccessful and returns false. Finally, if the put did not find the targeted key k, it creates a new node *node* with key k and value v that is linked to next (line 59) and inserts it physically using a CAS (line 60) as depicted in Figure 2(a)

The reason why nodes are logically deleted before being physically removed, as depicted in Figure 2(b), is to minimize contention. The delete function (Algorithm 2, lines 65–89) marks a *node* as logically deleted by setting its value to $\perp$. A separate *maintenance thread* is responsible for traversing the bottom level of the skip list to clean up the deleted nodes as described in Section 3.5. The delete executes like the put down to line 79 as it also traverses the structure from top to bottom (line 66) and backtracks to help_remove the deleted nodes at line 74. It checks whether a key is absent at line 77 or if its node is logically or physically deleted at line 82 in which case it returns false. Otherwise, this function logically deletes the node using a CAS to mark it at line 84. A heuristic (line 85) helps deciding whether the delete should help removing. This happens only when the ratio of logically deleted nodes over non-logically deleted nodes (as communicated by the background thread) reaches 3 after what it pays off. Whether it helps physically removing or not, the delete returns true if the node was logically deleted.

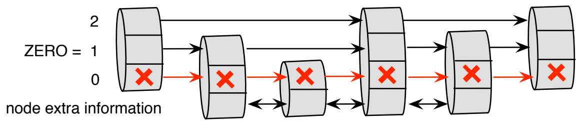
3.4 Wheels instead of towers

The wheel size is adjusted without having to mutate a pointer, simply by over provisioning a static array and using modulo arithmetic to adjust the mutable levels. The modulo arithmetic guarantees that increasing the index past the end of an array wraps around to the beginning of the array. This allows lowering to be done in constant-time by simply increasing the index of the lowest logical level of all the arrays (cf. Figure 3) without incurring contention. A global variable ZERO is used to keep track of the current lowest level (Alg. 1, line 17), and when a lowering occurs the lowest index level is invalidated by incrementing the ZERO variable (Alg. 3, line 102). This causes other threads to stop traversing the index levels before they reach the previous lowest level of pointers. If ZERO is increased above the length of the array this will not compromise the program since the arrays are accessed with modulo arithmetic.

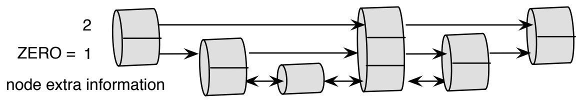
To illustrate how wheels improve locality of reference, consider line 22 of Algorithm 1, which involves testing to see if the current node *next* in the traversal has a key greater than the search key. If the wheels were represented using distinct objects, then $next.k$ at line 22 would need to be changed to $next.node.k$, reflecting the fact that the key-value information is being stored in a distinct node object from the index *next* references. This extra layer of indirection can hurt performance of the skip list, especially since this redirects all traversals.



(a) ZERO has value 0 when the lowering starts by incrementing it



(b) The lowest index item level gets ignored by new thread traversals



(c) Eventually the structure gets lowered and all wheels are shortened

Figure 3: Constant-time lowering of levels with ZERO increment (a) and (b), the bottommost level, which represents where the node extra information ($\langle k, v \rangle$, *prev*, *next*, *marker*, *level*) is located, is garbage collected later on

3.5 Background thread

The background (or maintenance) thread code is described in Algorithm 3. It executes a loop (lines 90–100) where it sleeps (line 94) for 50 microseconds (a parameter that showed good performance but could be easily tuned to the application). The maintenance thread raises wheels of non-deleted nodes by calling the raise_bottom_level function (line 95) similarly to [9], except that it applies to wheels instead of towers, as depicted in Figure 4. This raise is compensated with a constant-time lowering specific to our algorithm. This periodic adaptation makes it unnecessary to use the traditional pseudo-random generators as the lowering and *raising* (the action of increasing the height of a tower) become deterministic.

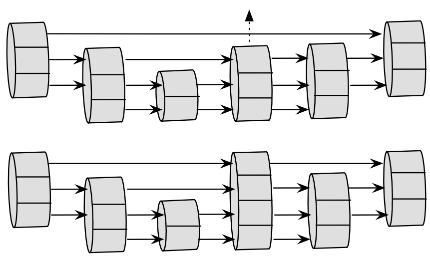


Figure 4: Raising consists of increasing the height of a tower

The lower_skiplist function (lines 101–102) discards, in constant-time, the entire bottom level of the skip list by simply changing the ZERO counter used in the modulo arithmetic (as depicted in Figure 3(a)), without blocking application threads. Note that the lower_skiplist is followed by a cleanup_bottom_level that takes linear time to reclaim the memory of the deleted level, however, all traversals starting after the ZERO increment ignores this level. The lower_skiplist function is called, with some heuristic, only if is_lowering_necessary returns true (line 100). The heuristic we chose in our experiments is if there are 10 times more wheels with height greater than 1 than bottom-level nodes. An interesting question is the choice of the ideal multiplying factor depending on n.

The raise_bottom_level (lines 103–111) also cleans up the skip list by removing the logically deleted nodes (line 108). After all logically deleted nodes are discarded, their wheels having been progressively lowered down to a single level, they are garbage collected using an epoch based memory reclamation algorithm discussed in Section 3.6. We omitted the definition of functions raise_node and raise_index_level that simply consist, for each level from bottom to top, of raising each node in the middle of three consecutive non-deleted nodes of the same height.

The help_remove function called by the remove function or by an application thread removes the deleted nodes. It actually only removes nodes that do not have any wheel successors. Nodes with wheels are removed

Algorithm 3 The rotating skip list: maintenance operations processed by the background thread b

```
90: background_loop()_b:
91:   node ← sl.head
92:   zero ← ZERO
93:   while not done do
94:     sleep(SLEEP_TIME)                          ▷ by default the delay is 50 µsec
95:     raise_bottom_level()
96:     for i ∈ {0..sl.head.level} do              ▷ for all levels
97:       raise_index_level(i)
98:     if is_lowering_necessary() then            ▷ if too unbalanced
99:       cleanup_bottom_level()                   ▷ garbage collect unused level
100:      lower_skiplist()                         ▷ lower the skip list

101: lower_skiplist()_b:                           ▷ constant-time lowering
102:   ZERO ← ZERO + 1 % wheel_capacity            ▷ ignore bottom index level

103: raise_bottom_level()_b:
104:   node ← sl.head
105:   next ← node.next
106:   while node ≠ ⊥ do
107:     if node.val = ⊥ then
108:       remove(prev, node)                      ▷ help with removals
109:     else if node.v ≠ node then
110:       raise_node(node)                        ▷ only raise non-deleted nodes
111:     node ← node.next

112: cleanup_bottom_level()_b:
113:   node ← sl.head
114:   zero ← ZERO
115:   while node ≠ ⊥ do                          ▷ up to the tail
116:     node.succs[(zero + 0) % max_levels]       ▷ nullify the wheels
117:     node.level ← node.level − 1              ▷ decrement this node's level
118:     node ← node.next                          ▷ continue with immediate next node

119: remove(prev, node)_b:
120:   if node.level = 1 then                     ▷ remove only short nodes
121:     CAS(&node.v, ⊥, node)
122:     if node.v = node then help_remove(prev, node)

123: help_remove(prev, node)_b:
124:   if node.v ≠ node ∧ node.marker then
125:     return                                    ▷ nothing to be done
126:   n ← node.next
127:   while n = ⊥ ∨ n.marker ≠ true do           ▷ till a marker succeeds node
128:     new ← new_marker(node, n)
129:     CAS(&node.next, n, new)
130:     n ← node.next
131:   if prev.next ≠ node ∨ prev.marker then
132:     return
133:   res ← CAS(&prev.next, node, n.next)         ▷ unlink node+marker
134:   if res then                                 ▷ free memory for node and marker
135:     delete_node(node)
136:     delete_node(n)
137:   prev_next ← prev.next
138:   if prev_next ≠ ⊥ then prev_next.prev ← prev  ▷ no need for accuracy
```

differently by first lowering their levels. Deleted wheels are simply removed later by the background thread within a help_remove during maintenance iteration (Alg. 3, lines 133–136). Note that at the end of the removal (line 138) the *prev* field can be updated without synchronization as it does not have to be set to the immediate previous node.

3.6 Memory reclamation

The memory reclamation (whose pseudocode is omitted here) of our rotating skip list is based on an epoch based garbage collection algorithm similar to the one used in Fraser's skip list with some differences. The garbage collector of Fraser's skip list is partitioned into sections responsible for managing nodes of a particular skip list level. Partitioning the memory manager like this means that requests to the memory manager regarding different skip list levels do not need to conflict with one another. In contrast, the rotating skip list

man.	freq.	#soc.	#cores	#thrds.	L1$I	L1$D	L2$	L3$
AMD	1.4GHz	4	64	64	16KiB	64KiB	2MiB	6MiB
Intel	2.1GHz	2	16	32	32KiB	32KiB	256KiB	20MiB

Table 2: The multicore configurations used in our experiments, with the manufacturer, the clock frequency, the total numbers of sockets, cores and threads, and the size of L1 instruction cache, L1 data cache, L2 and L3 caches

does no such partitioning of memory management responsibilities, since the rotating skip list uses only one node size for all list elements regardless of their level. This increases the probability of contention in the memory reclamation module when a large number of threads issue memory requests simultaneously.

4 Experimental Settings

In this section we describe the multicore machines and the benchmarks used in our experiments as well as the 7 data structure algorithms we compare our rotating skip list against.

Multicore machines. We used two different multicore machines to validate our results, one with 2 8-way Intel Xeon E5-2450 processors with hyperthreading (32 ways) running Fedora 18 and gcc 4.7.2 and another with 4 16-way AMD Opteron 6378 processors (64 ways) running Ubuntu 12.04.4 LTS, OpenJDK 64-bit server VM IcedTea 2.5.3 and gcc 4.8.1. Both share the common x86_64 architectures and have cache line size of 64 bytes. The two multicore configurations are depicted in Table 2.

The Intel and AMD multicores use different cache coherence protocols and offer different ratios of cache latencies over memory latencies, which impact the performance of multi-threaded applications. The Intel machine uses the QuickPath Interconnect for the MESIF cache coherence protocol to access the distant socket L3 cache faster than the memory and the latency of access to the L1D cache is 4 cycles, to the L2 cache is 10 cycles and to the L3 cache is between 45 and 300 cycles depending on whether it is a local or a remote access [21]. The latency of access to DRAM is from 120 to 400 cycles. The AMD machine uses HyperTransfer with a MOESI cache coherence protocol for inter-socket communication, it has a slightly lower latency to access L1 and L2 caches and a slightly higher latency to access the memory [6].

Data structures and synchronization techniques. To compare the performance of the rotating skip list, we implemented 6 additional data structures, including 4 skip lists and 2 balanced trees. Two of the skip lists use the same synchronization technique as our rotating skip list, they are non-blocking thanks to the exclusive use of CAS for synchronization. We also tested data structures using transactions, RCU (read-copy-udpate), and locks. Note that we used both pthread mutexes and spinlocks but report only the best performance (obtained with spinlocks as mutexes induce context switches that predominate in-memory workload latencies), we also used software transactions due to the limited L1 data cache size of 32KiB and the recently found bug[1] of the Intel Haswell HTM.

1. Fraser's skip list is the publicly available library [14] developed in C by Fraser. It includes several optimisations compared to the algorithm presented in his thesis [15]. We kept this library unchanged for the sake of fair comparison.

2. Crain's no hotspots skip list is the algorithm presented recently [9]. As the original algorithm was developed in Java, we ported it in C while keeping the design as close as possible to the original pseudocode. In particular, we present the performance of the algorithm without any particular garbage collector. Memory reclamation tends to be costly in our benchmarks.

3. The Optimistic Skip List algorithm [18] exploits the logical deletion technique to allow traversals to proceed without locking but acquires locks locally to insert or delete a node. Before returning, an access must validate that the value seen is not logically deleted and that the position is correct. It may restart from the beginning of the structure if not successful. We ported the pseudocode in C and we implemented an exponential backoff contention manager to cope with infinite restarts.

4. Our transaction-based skip list uses a classic skip list [28] whose accesses were protected using elastic transactions as offered by the $\mathscr{E}$-STM software transactional library [12]. Elastic transaction is a relaxed transaction model that was shown particularly promising on pointer-based structures, like skip list.

[1] http://www.theregister.co.uk/2014/08/14/intel_disables_hot_new_tsx_tech_ in_early_broadwells_and_haswells/.

5. The Speculation-Friendly Binary Search Tree [8] is a binary search tree data structure that exports a key-value store interface and uses transactions for synchronization. We kept the code from the Sycnchrobench benchmark-suite and used the recommended $\mathscr{E}$-STM software library to synchronize it. This tree algorithm was shown efficient on a travel reservation database application.

6. The Citrus tree [1] is, as far as we know, the only RCU-based tree to allow concurrent update operations. It uses Read-Copy-Update synchronization technique (RCU) to achieve fast wait-free read-only accesses [17]. Using RCU, updates produce a copy of the data they write to let read-only accesses execute uninterrupted without acquiring locks. We use the implementation provided by the authors and as they recommended we employ an external memory allocator library designed for concurrent environment. We used Thread-Caching Malloc[2] as it is the one used with the fast lock-free binary search tree.

7. We chose the Java implementation provided by the authors of an efficient lock-free tree [11]. Although this tree does not rebalance, we only tested it in scenarios where it should perform best: where inserted and removed values are uniformly distributed so that the tree does not get unbalanced.

Benchmarks. We have integrated all 7 logarithmic data structures in the Synchrobench benchmark suite [16]: Synchrobench is an open source benchmark suite written in C/C++ and Java to evaluate data structures using multiple synchronization techniques in a multicore environment. The code of the Rotating skip list is thus publicly available online at `https://github.com/gramoli/synchrobench`.

We parameterized Synchrobench with effective update ratios of 0%, 10% and 30%, various threads counts in $\{1,4,8,16,32,48,56,64\}$ on the AMD machine and $\{1,4,8,12,16,20,24,28,32\}$ on the Intel machine, two data structure initial sizes of 2^{10} and 2^{16} and we kept the range of keys twice as big as the initial size to maintain the size constant in expectation all along each experiment. Each point on the graphs of Section 5 displays error bars for a 95% confidence interval (assuming a normal distribution of values) and consists of the average of 20 runs of 5 seconds each. Each point on the graphs of Section 6 is averaged over 10 runs of 5 seconds each and do not include error bars. Our reports of cache miss rates are taken from different runs on AMD from those used for throughput measures, to avoid impacts from the cache profiling on performance: each miss rate figure is the overall value observed from the total access pattern during 8 runs, where one run has each value of thread count from the set $\{1,4,8,16,32,48,56,64\}$.

5 Evaluating Non-Blocking Skip Lists

We evaluated the performance of each skip list in terms of throughput. Throughput has been the crucial metric to assess the performance of databases as the number of transactions per second. In the context of the key value store, each transaction is simply an operation, hence we depict the throughput in millions (M) of operations per second.

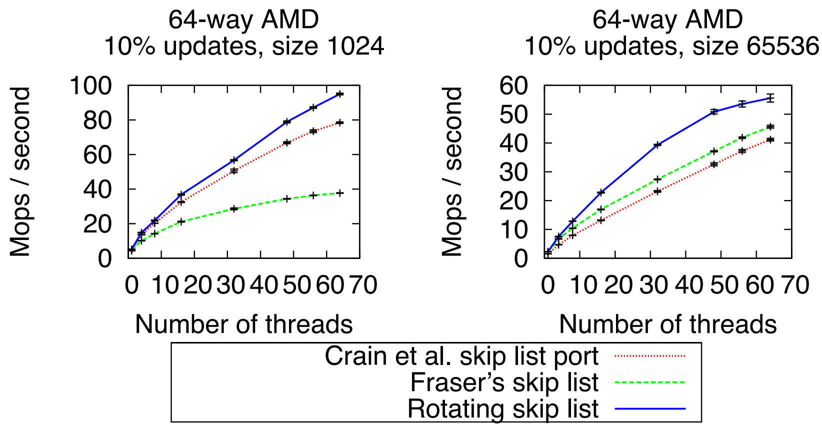


Figure 5: The rotating skip list outperforms both the Crain's and Fraser's skip lists at all thread count under 10% update

[2]http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

5.1 Peak throughput

Figure 5 depicts the performance of the rotating skip list in the exact same settings as the one reported in Figure 1. The performance are given for a small skip list ($2^{10} = 1024$ elements) and a large skip list ($2^{16} = 65536$ elements) experiencing the commonly used update:readonly ratio of 1:9. In both workloads the rotating skip list achieves higher performance than the two existing skip lists.

While Fraser's skip list was proposed a decade ago [15], it is often considered as the most efficient skip list implementation. Recent skip list implementations, although very close in performance, either get outperformed when the number of threads grows [10] or under simple read-only workloads [2]. As we show in detail below, the higher performance of the rotating skip list stems from the use of wheels that reduce hotspots and are cache efficient.

5.2 Tolerance to contention

We increase the update accesses to 30% and 100% of the workload to see how contention is handled. Perhaps the most interesting result is that the rotating skip list outperforms substantially Fraser's.

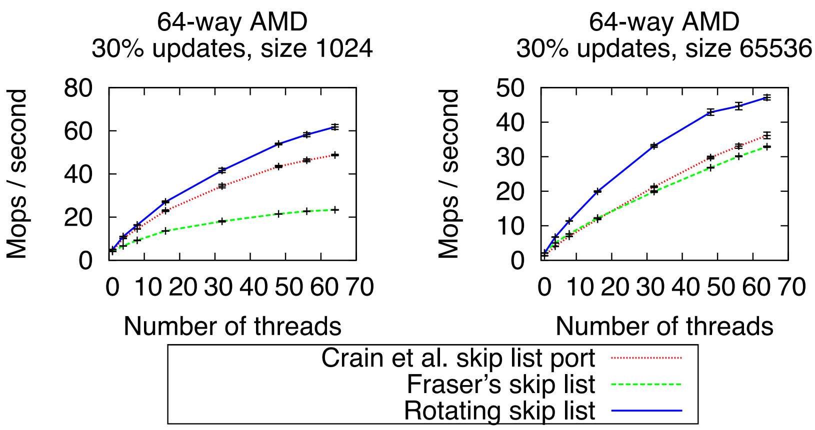


Figure 6: The rotating skip list does not suffer from contention hotspots

Figure 6 indicates that the rotating skip list outperforms Fraser's by a multiplying factory of up to 2.6. This improvement stems from the ability of the rotating skip list to deal with contention effectively and is more visible for relatively small data structures (2^{10} elements) where the probability of contending increases. Only does the bottom level of the list need synchronization, which reduces significantly the number of CAS necessary to insert or delete a node without data races. In addition, the higher levels of traditional skip lists are known to be hotspots and they are more likely traversed during any operation. Using CAS on these top levels, as it is the case in Fraser's (and traditional non-blocking skip lists), tends to slow down the traversals by introducing implicit memory barriers.

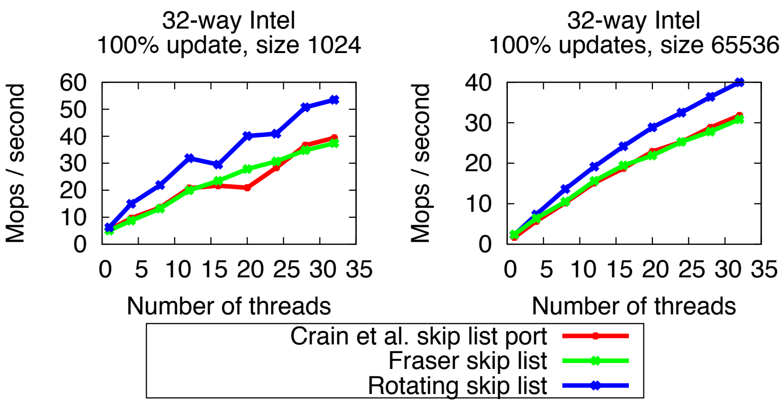


Figure 7: Intel measurements to evaluate performance under high contention

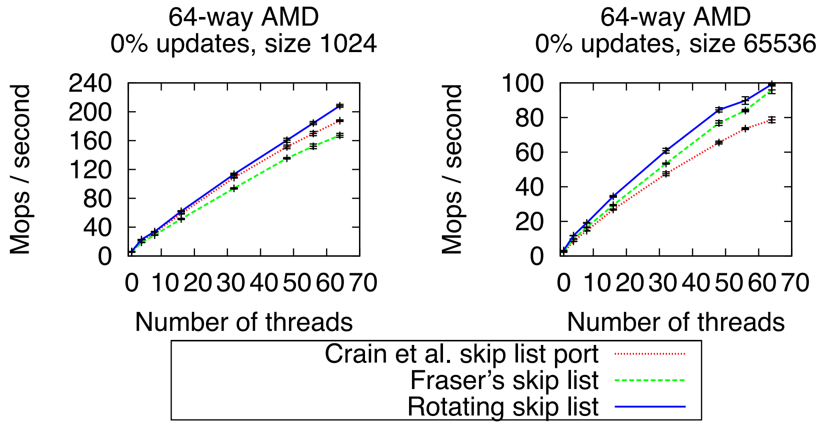


Figure 8: The locality of reference of the rotating skip list boosts read-only workloads

Finally, we also tested the performance of the skip lists under higher contention. We set the attempted update ratio to 100%, meaning that 50% of the operations would effectively update the data structures whereas 50% of them would fail and return unsuccessfully without updating the structure. Figure 7 depicts the performance obtained with the Rotating skip list, the No Hotspot skip list (Crain et al. port) and Fraser's skip list on the Intel machine for 2^{10} elements (left) and 2^{16} elements (right). We observe that, as expected, the rotating skip list performs significantly better than Fraser's and the port of the No hotspot skip list on Intel, which confirms our aforementioned observations on AMD.

5.3 Locality of reference

We also performed comparative evaluations in settings with no contention at all. Figure 8 indicates the performance of the three skip lists when there are no update operations. Fraser's skip list is known to be particularly efficient in such settings [2], but our rotating skip list appears to be more efficient both on small (2^{10} elements) and large (2^{16} elements) datasets.

This improvement could be due to not having to synchronize top levels of the structure, however, Crain's skip list presents lower performance than the rotating one. As no remove operations execute, this difference is expressed by the way nodes are represented in memory. The rotating skip list represents a wheel as an array so that nodes that are above each other get recorded in contiguous memory locations. The main advantage is that multiple nodes can be fetched from memory at once, when a single cache line is stored in the cache. Later, when the same thread traverses down the structure, accessing a lower node will likely produce a cache hit due to spatial locality. Instead, Crain's skip list uses a linked structure to represent towers, so that traversing down the list requires to fetch disjoint memory locations that will not likely be in the cache already.

5.4 Cache miss rate

To confirm that the rotating skip list leverages caches more effectively than Crain's skip list, we measured the amount of cache misses. Table 3 depicts the number of cache misses (in parentheses), and the cache miss rate (that is, the proportion of cache misses out of all accesses) observed while running Crain's no hotspot skip list and the rotating skip list on a key-value store with 2^{16} elements. We can see that the rotating skip list decreases the cache miss rate substantially when compared to the no hot spot skip list, under each update rate. This confirms the impact of cache efficiency on performance we conjectured in the former sections.

update	0%	10%	30%
Crain port	(29,834K) 0.39‰	(35,457K) 0.83‰	(41,027K) 1.07‰
rotating	(28,061K) 0.29‰	(26,258K) 0.37‰	(24,094K) 0.38‰

Table 3: Compared to Crain's skip list, the rotating skip list limits the cache miss rate on datasets of size 2^{16} (absolute cache miss numbers in parentheses)

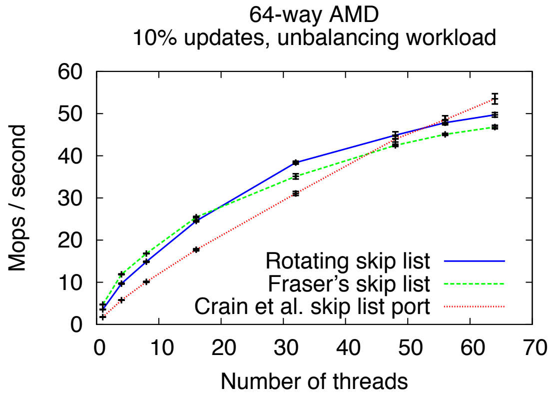


Figure 9: The rotating skip list rebalances effectively under highly skewed workloads

5.5 Stressing the maintenance thread

To reduce contention it is important that a single thread does the maintenance. To stress this aspect, we measure the performance under a skewed workload that unbalances the structure. We have extended Synchrobench with an additional U parameter that initially populates the structure with (1024) keys that are within a subrange ($[0,1024]$) of the admitted range ($[0,32768]$); application threads then insert/delete with probability 10% by picking uniformly a key within the range $[0,32768]$. The impact is that most inserts executed during the experiment fall to the right of the pre-existing towers, unbalancing the skip list.

Figure 9 depicts the results: a single maintenance thread in the rotating skip list is enough to raise the new wheels inserted by the application threads. In particular, the rotating skip list provides performance superior to Fraser's starting at 32 threads. Although all three skip lists give similar performance, Crain's gives slightly higher performance for more than 56 threads, probably because it is the only one that does not support memory reclamation (cf. Section 8 for details on the restructuring effectiveness).

6 Extra Synchronizations and Structures

We also compared the performance of the rotating skip list against a non-blocking tree, a RCU-based tree and a transaction-based tree, as well as lock-based and transaction-based skip lists.

6.1 Synchronization techniques

Our lock-based skip list implements the Optimistic skip list algorithm [18]. Our transaction-based skip list uses a classic skip list [28] whose accesses were protected using elastic transactions as offered by the $\mathscr{E}$-STM software transactional library. (As mentioned previously, due to the limited L1 (data) cache size (32KiB) of the Intel Haswell processor and the recent bug finding TSX we decided not to use hardware transactions.) Elastic transaction is a relaxed transaction model that was shown particularly efficient on pointer-based structures [12], like skip lists.

Figure 10 depicts the performance of the lock-based skip list, the transaction-based skip list, Fraser's and the rotating one on the Intel machine. (We omitted the skip list port from Crain et al. as it has no memory reclamation.) First, we can see that the rotating skip list outperforms the others. It also is slightly faster on Intel than on AMD at equivalent thread counts (e.g., 32), this is due to the 33% higher clock frequency of the Intel machine than the AMD's (2.1GHz vs. 1.4Ghz) but performance degrades slightly under high contention (1024 elements) between 16 and 24 threads on Intel as the second socket starts inducing additional off-chip traffic.

The lock-based skip list performance stops scaling after 8 threads on 1024 elements and after 12 threads on 65536 elements. The reason is that the data structure suffers from contention, especially where the number of elements is smaller, increasing the chance of having multiple threads conflicting on the same nodes. When such a conflict occurs, the validation fails which typically restarts the operation from the beginning. The transaction-based skip list does not scale either on 1024 elements due to the contention. Since it exploits the elastic transaction model that is relaxed, its performance scale above the lock-based skip list on 65536 elements.

Despite the TSX bug recently announced, we could have tested hardware transactions on an 8-way Haswell processors, however, recent performance results of a 1024-sized red-black tree with 10% up-

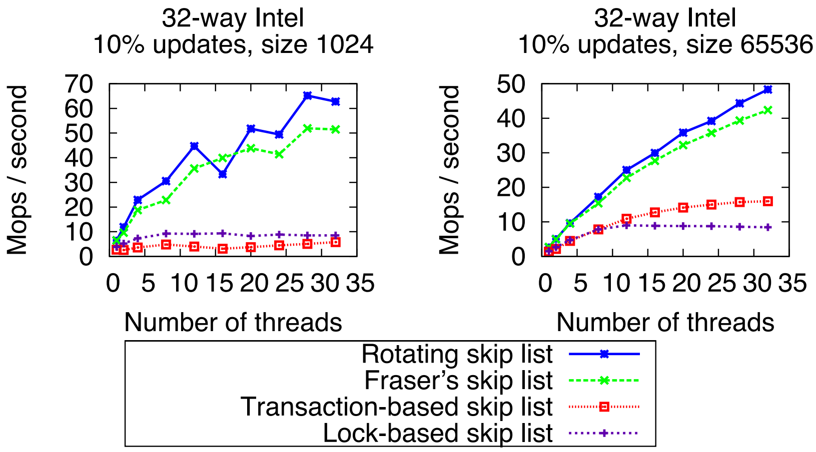


Figure 10: Intel measurements to compare the performance against other synchronization techniques

dates [25] already indicate a performance peak around 28 M operations/second, which is 12% slower than the throughput of the rotating skip list on our Intel Xeon machine running only 8 threads.

6.2 Balanced trees

A balanced tree is a data structure, found in various applications, that is more popular than skip list but has a comparable asymptotic complexity. To illustrate the applicability of our rotating skip list we compared its performance to three recent balanced tree data structures.

We benchmarked the lock-free tree that does not rebalance [11] as a baseline because it should perform well when the workload is balanced (as in our case). We also benchmarked the Speculation-Friendly Binary Search Tree [8] that uses optimistic concurrency control and delay the rebalancing under contention peaks. Finally, we benchmarked the recent Citrus Tree [1], a balanced tree that exploits the Read-Copy-Update synchronization technique (RCU) to achieve fast wait-free read-only accesses and concurrent updates.

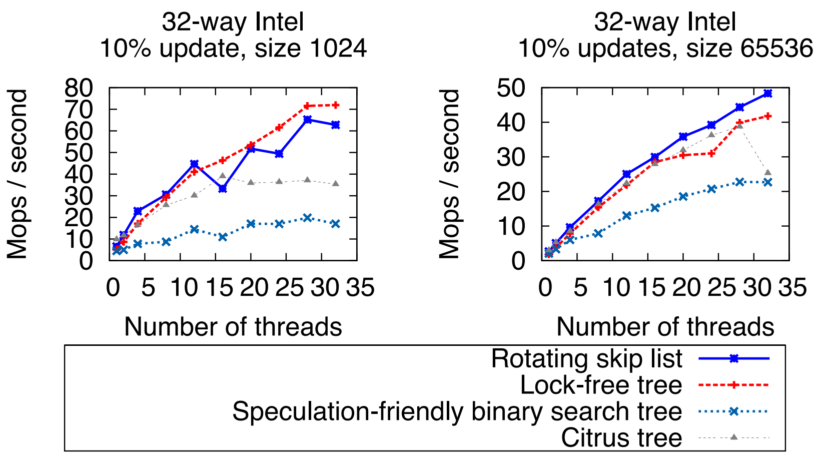


Figure 11: Intel measurements to compare the performance against balanced tree data structures

Figure 11 depicts the performance we obtained when running the rotating skip list, the fast lock-free binary search tree, the speculation-friendly tree and the Citrus tree on the Intel machine. A first observation is that the rotating skip list is generally the fastest among these data structures. Another observation is that the performance of the Lock-Free Tree never scales to 32 threads—at the time of writing we are still exploring the code from the authors to understand these fluctuations. Despite its lack of garbage collector, the Citrus tree has lower performance than the two others probably due to the global counter used in the user-level RCU library. Finally, the Speculation-Friendly tree scales up to 32 threads but remains slower than the other, most likely due to the extra bookkeeping of the software TM library.

7 Correctness and Progress

Our key-value store implementation is linearizable [20]. This property, also known as atomicity, ensures that during any concurrent execution each operation appears as if it was executed at some indivisible point of the execution between its invocation and response. We refer to this indivisible point of a particular operation (or execution of a function) as its *linearization point*. The get function proceeds by simply reading the data structure as it never updates the data structure directly: its help_remove call triggers the removal executed by the maintenance thread. The get function is guaranteed to terminate once it evaluates the precondition at line 38 to true otherwise it continues traversing the structure. The linearization point of its execution, regardless of whether it returns *val* or $\perp$, is at line 38.

There are three cases to consider to identify the linearization point of an execution of the put function. First the put executes the same traversal to the bottom level as the one of the get in which it never updates the structure. If the execution of the $\text{put}(k, v)$ actually finds a node with key k then it has to check whether it is logically deleted, but we know that either way from this point, the put will return without iterating once more in the while loop. First, if the node with key k has its value set to $\perp$ indicated that it has been logically deleted, then the put inserts it logically by executing a CAS that ensures that no concurrent thread can logically insert it concurrently. This CAS, at line 56 is actually the linearization point of the execution of put that inserts logically a node. Second, if the node with key k has its value set to another value $v' \neq \perp$ then the node is not logically deleted, and put returns false. In this case the linearization point is at the point where *val* was read for the last time at line 45. Third, if the node is not already in the key-value store then it is inserted with a CAS at line 60, which is the linearization point of this execution of put.

There are four cases to consider for the delete execution. The code is similar to the code of the get and put down to line 79. First, if the predicate evaluates to true at line 79, then the delete returns false at line 88 because the key-value pair to be deleted is not present. Line 68, which is the last point at which *val* was read, is the linearization point of this execution of the delete. Second, if the key k does not exist, then the delete returns false at line 77, in this case the linearization point is at line 77 where k is lastly read. Third, if the key-value pair *node* is already logically deleted ($val = \perp$) or removed ($node = val$), which is checked at line 82, then the delete returns false and its linearization point is at line 81. Finally, if the delete returns true then this means that the key was found and the corresponding node has successfully been deleted. Note that if the CAS fails, then the loop restarts from the point where the bottom level was reached. The linearization point of a successful delete is at line 84 where the CAS occurs. It is easy to see that any execution involving three functions get, put and delete occurs as if it was occurring at some indivisible point between their invocation and response as each of these functions accepts a linearization point that refers to an atomic event that can only occur after the function is invoked and before it returns.

Our key-value store is non-blocking. It only uses CAS as a synchronization technique for updating and no synchronization technique during a get. The only way for an operation to not exit a loop is to have its CAS interfere with another CAS executed by another thread on the same memory location. Note that this guarantees that each time a CAS fails, another succeeds and thus the whole system always makes progress.

8 Time and Space Complexity

The time complexity of a skip list depends on the distribution of nodes per level. For example, if there are twice as many nodes (uniformly balanced) at one level ℓ than at its upper level $\ell + 1$, a lookup halves (in expectation) the number of candidate nodes each time it goes down a level, hence leading to an asymptotic complexity that is logarithmic (in base 2) in the number of nodes the structure contains. Our algorithm does not use any pseudo-random number generator. Even though the skewness in the placement of high wheels negligibly affects its performance (cf. Section 5.5) it remains unclear whether its deterministic adjustment is fast enough to rebalance under high contention.

To answer this question, we plotted the number of nodes per level after having the 32 cores of our Intel machine ran [10%..50%] of updates during 20 seconds. At the end of these 20 seconds, without waiting any delay, we measured the number of nodes present at each level, as depicted in Figure 12 with a scale on

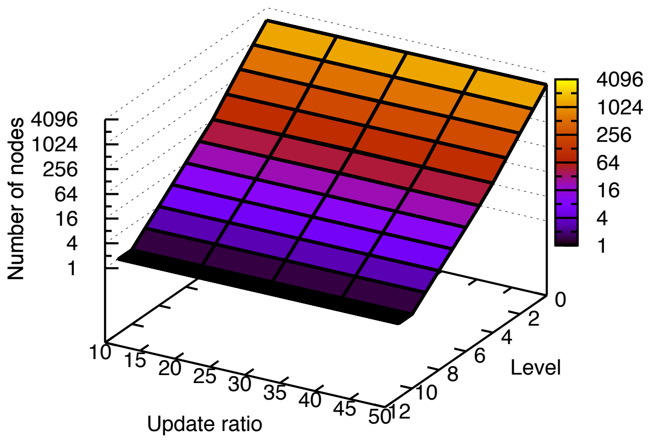


Figure 12: The distribution of node per level indicates that there are always about twice as many nodes at level ℓ than at level $\ell + 1$ regardless of the update ratio

the vertical axis that is logarithmic in base 2. Under reasonably low contention (10%) the logarithm of the number of nodes at level ℓ seems linearly proportional to $maxlevel - \ell$, indicating that there are about twice as many nodes at level ℓ than at level $\ell + 1$. We also observe that the contention has negligible impact on this distribution, indicating that our restructuring maintains effectively the distribution of node per level to keep time complexity close to $\log_2 n$.

One drawback of our skip list is its space complexity. This problem is rather due to a fixed wheel capacity than the number of logical nodes, which is typically $O(n \log n)$ in realistic workloads. While it suffices to increase the wheel capacity by k to allocate 2^k more elements, the current implementation requires to allocate memory at start time. One could extend the current implementation to bound the space complexity by implementing a dynamic resize. A universal construction could be used to avoid blocking [19] while a resize that use locks may be used as long as it does not block the get/put/delete operations.

9 Related Work

Several lock-free data structures were recently observed faster than their lock-based counterpart on multi-cores [16]. Fraser [15] proposed a method for constructing a non-blocking binary search tree using a multi-word CAS primitive. Ellen et al. proposed the first non-blocking binary search tree with just single-word CAS [11] that we have evaluated here. Braginsky and Petrank [3] proposed a non-blocking B+-tree implemented using just single-word CAS operations. Mao et al. [24] combined tries with B+-trees to offer non-blocking lookups that reduce cache-misses in persistent storage.

Levandoski et al. [22] proposed the BW-tree, a cache-friendly version of a B+-tree, implemented using CAS to reduce blocking to I/O. The authors report that their implementation outperforms a latch-free skip list implementation, however, the skip list does not include the aforementioned optimizations and at the time of writing none of these proprietary implementations are available for comparison. Chatterjee et al. [5] proposed a lock-free binary search tree, but we are not aware of any implementation.

Skip lists are considered simple alternatives to non-blocking balanced trees. Fraser proposed a C-based implementation of a non-blocking skip list [15] relying exclusively on CAS for synchronization with an epoch based memory reclamation technique. Doug Lea integrated a variant of Fraser's skip list in the JDK since version 1.6 but it was reported $2\times$ slower than the no hotspot skip list we evaluated here [9]. Sundell and Tsigas [29] proposed a non-blocking skip list to implement a dictionary abstraction using CAS, test-and-set and fetch-and-add. Fomitchev and Ruppert [13] proposed a non-blocking skip list, however, they do not offer an implementation of their approach. Recently, a shallow skip list was combined with a hash table to store high level nodes [27] but it uses a double-wide double-word-compare-single-swap.

Crain et al. proposed the no hotspots skip list [9] but it requires an implicit garbage collector and does not offer constant-time lowering. The way the no hotspots skip list raises towers is similar to the way the rotating skip list raises wheels. A dedicated thread periodically sets up an array containing the first element of every list level. If three consecutive towers have the height of the level being traversed, the height of the middle tower is incremented.

Other synchronization techniques were used to implement logarithmic data structures. Transactional memory (TM) was used to transactionalize accesses to logarithmic data structures, like AVL and red-black trees [4]. Crain et al. split tree accesses into multiple smaller transactions to reduce the overhead induced by conflicts in the speculation-friendly binary search tree [8]. Dragojević and Harris exploited mini-transactions to speedup a skip list [10] whereas Avni et al. used TM to implement snapshot [2]. The peak performance of these two skip lists is similar to Fraser's. Read-copy-update (RCU) became popular in the linux kernel. The Bonsai tree [7] exploits RCU but does not offer concurrent updates. Arbel and Attiya proposed RCU-based concurrent updates [1], however, it is slower than our skip list.

10 Conclusion

We proposed new algorithmic design choices to implement efficient logarithmic data structures for modern multicores by favoring cache hits and limiting contention. We illustrated these choices by devising a novel data structure, the Rotating skip list that uses rotating wheels instead of towers. The Rotating skip list outperforms most recent concurrent trees and skip lists on commodity multicore machines, indicating that it could be the structure of choice to implement key-value stores.

Acknowledgments

This research was supported under Australian Research Council's Discovery Projects funding scheme (project number 160104801) entitled "Data Structures for Multi-Core". Vincent Gramoli is the recipient of the Australian Research Council Discovery International Award.

References

[1] M. Arbel and H. Attiya. Concurrent updates with RCU: Search tree as an example. In *PODC*, pages 196–205, 2014.

[2] H. Avni, N. Shavit, and A. Suissa. Leaplist: Lessons learned in designing TM-supported range queries. In *PODC*, 2013.

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

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, September 2008.

[5] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. In *PODC*, pages 322–331, 2014.

[6] J. Chen and W. Watson III. Multithreading performance on commodity multi-core processors. http://usqcd.jlab.org/usqcd-docs/qmt/multicoretalk.pdf.

[7] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *ASPLOS*, pages 199–210, 2012.

[8] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *PPoPP*, pages 161–170, 2012.

[9] T. Crain, V. Gramoli, and M. Raynal. No hot spot non-blocking skip list. In *ICDCS*, 2013.

[10] A. Dragojević and T. Harris. STM in the small: trading generality for performance in software transactional memory. In *EuroSys*, pages 1–14, 2012.

[11] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *PODC*, pages 131–140, 2010.

[12] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, pages 93–107. Springer-Verlag, 2009.

[13] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59. ACM, 2004.

[14] K. Fraser. C implementation of Fraser's non-blocking skip list. http://www.cl.cam.ac.uk/research/srg/netos/lock-free/.

[15] K. Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003., 2004.

[16] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *PPoPP*, 2015.

[17] D. Guniguntala, P. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Systems Journal*, 47(2):221–236, 2008.

[18] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *SIROCCO*, pages 124–138, 2007.

[19] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[20] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990.

[21] Intel. Performance analysis guide - Intel developer zone, 2008. http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[22] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *ICDE*, pages 302–313, 2013.

[23] M. M. Maged. The balancing act of choosing nonblocking features. *Commun. ACM*, 56(9):46–53, Sept. 2013.

[24] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, pages 183–196, 2012.

[25] A. Matveev and N. Shavit. Reduced hardware NOrec: An opaque obstruction-free and privatizing HyTM. In *TRANSACT*, 2014.

[26] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[27] R. Oshman and N. Shavit. The SkipTrie: low-depth concurrent search without rebalancing. In *PODC*, pages 23–32, 2013.

[28] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[29] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *SAC*, pages 1438–1445. ACM, 2004.

[30] B. Wicht. Binary trees implementations comparison for multicore programming. Technical report, Switzerland HES-SO University of applied science, 2012.