Logarithmic Data Structures for Multicores

[Unpublished draft]

Ian Dick Alan Fekete Vincent Gramoli
University of Sydney
firstname.name@sydney.edu.au

Abstract

As multi-core and many-core machines become more common, there is a growing need for data structures that scale well with the number of concurrent threads of execution. Two factors of scalability limitations separately investigated by researchers are the contention of multiple threads accessing the same locations of a structure and the overhead of off-chip traffic.

Though the ideas of alleviating contention and improving memory locality appear elsewhere, we present the first data structure to combine them. This, so called, rotating skip list is a deterministic data structure with a lightweight restructuring process that trades towers for wheels to provide memory locality and maximise cache hits, and avoids contention hotspots.

We compare the performance of the rotating skip list against 6 other logarithmic data structures on 2 multicore machines from AMD and Intel using 4 different synchronization techniques: locks, compare-and-swap, read-copy-update and transactions. The results show that the rotating skip list is the fastest skip list to date.

Categories and Subject Descriptors D.1. Programming Techniques [*Concurrent Programming*]: Parallel programming

General Terms Algorithms, Performance

Keywords contention; cache-friendliness; non-blocking

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. These multicore machines adopt a concurrent execution model where multiple threads exploit computational resources of cores while accessing in-memory shared data. To continue the pace of increasing software performance, the throughput has to scale with the level of concurrency.

The crux of this scalability challenge is to revise decades of research on data structures to let multiple threads access efficiently shared data structures. While effort was devoted to reduce their asymptotic complexity to maintain high performance as the amount of data grows, few aimed to reduce their inherent contention so as to maintain high performance as the thread count grows. In particular, structures that can be accessed by multiple threads concurrently, must avoid the *contention* arising when a growing amount of threads try to update the same data simultaneously.

This contention issue is particularly visible on constant access time data structures [29]. As the amount of potential concurrent accesses on commodity hardware typically outnumbers the amount of accessed locations on these structures, an effective idea is to serialise accesses to remove the bottleneck. The serialising technique is however limited by Amdhal's law. Other data structures, like balanced trees or skip lists, provide a higher level of parallelism while providing reasonably low asymptotic complexity, typically logarithmic. Yet, these *logarithmic data structures* must satisfy global balancing constraints that induce contention. For example, when a tree gets unbalanced a global rebalancing is needed, potentially creating a contention hotspot at its root.

A second vital design goal is to minimise the off-chip traffic produced by the hardware to maintain data coherence. Threads have to use a synchronisation technique to avoid interfering with one another, typically when one tried to update the data that another is accessing. Mutual exclusion is a popular synchronisation technique, however, the related lock metadata management induces cache traffic which might affect scalability. Some locks, like test-and-set spinlocks are prone to the "bouncing problem" where acquiring a lock invalidates the cache of all threads reading the lock as they are waiting for its release. Although some recommend the use of more scalable locks [17], there is a growing interest in non-blocking operations based on condition variables to design scalable data structures [21–23].

In this paper, we tackle the fundamental problem of designing logarithmic data structures for multicore. We explore several state-of-the-art logarithmic data structures, including skip lists and balanced trees. A *skip list* is a probabilis-

tic data structure that stores data into nodes complemented with linked towers that provide shortcuts to traverse the data structure in expected logarithmic time [28]. Each insertion (resp. removal) of a node requires to link (resp. unlink) its complete tower to (resp. from) the structure. A *balanced tree* guarantees logarithmic access time deterministically but requires that among all the paths from the root to a leaf, the length of the longest path is not far from the length of the shortest path.

We propose the rotating skip list, aimed at a cache-friendly C implementation that does not experience hotspots. In short, it uses neither locks nor object node indirection, so as to minimise cache traffic and maximise memory locality, and it uses *wheels* instead of towers of nodes to adjust the structure sequentially. In its most common representation, including in the Java Development Kit, skip list towers are represented as a set of node objects referencing each other. Accessing multiple pointers often requires as many accesses to the memory, and thus pointers can lead to poor cache behaviour. An alternative is to represent each tower as an array to exploit memory locality to minimise traffic by maximising cache hits. The drawback of arrays is the cost of adding (resp. removing) an element to (resp. from) them.

The wheels of our rotating skip list are the key to avoiding both hotspots and indirections of node objects: the skip list is divided into a doubly linked list at the bottom level and a wheel on top of each node implemented as a rotating buffer array. A maintenance thread is dedicated to adapt the structure while the other threads can only modify the bottom-most level of the skip list. The wheel size does not change, instead modulo arithmetic is used to adjust a ZERO marker that wraps around the buffer to indicate the lowest level node of the wheel. Only one access to memory is necessary to access multiple times the same wheel during a traversal, since the subsequent accesses produce cache hits. In addition, as the linked list is probably the most concurrency-friendly data structure [31] its bottom list is well-suited to support a high number of concurrent insertions and removals.

More precisely, we make the following contributions:

- We propose a novel data structure, the *rotating skip list*, that combines advantages from both a tree and a skip list. Its insertions (resp. removals) simply modify the localized bottom-most nodes and do not link (resp. unlink) towers. It is deterministic and does not need rebalancing thanks to a separate maintenance thread that check and adjusts the structure in the background. It is cache-friendly and it avoids contention hotspots at the top levels.
- To illustrate the usefulness of our multicore data structure, we implemented a key-value store abstraction that we ported on two commodity multicore machines with different cache coherence protocols: a 64-way AMD Opteron and a 32-way Intel Xeon. A key-value store is a popular abstraction in in-memory database system to map a key to a value, traditionally implemented with trees it has been implemented more recently with skip lists (e.g., MemSQL[1], ArangoDB[2]).
- We compare the performance of the rotating skip list against the two most efficient concurrent skip list algorithms: Fraser's skip list [13] and the skip list from Crain et al. [7]. We show that our algorithm has an advantage in the presence of high contention, even though we observed lower performance when the distribution of update requests was highly skewed towards a specific key subrange. As far as we know, the rotating skip list is also the first to outperform these skip lists on read-only workloads, hence achieving unprecedented peak performance.
- Finally, we compare logarithmic data structures with four synchronisation techniques using the synchrobench micro-benchmark suite [10]: locks, condition variables (e.g., compare-and-swap/CAS), read-copy-update (RCU) and transactions. We first compare the CAS-based rotating skip list against the lock-based optimistic skip list [16], the CAS-based Fraser's skip list [13], the CAS-based skip list from Crain et al. [7] and the transaction-based skip list [10]. Then we compare its performance against two balanced tree algorithms used for operating systems and database applications, the RCU/lock-based bonsai tree [5] and the transaction-based speculation-friendly tree [6].

In Section 2, we state the problem and in Section 3 we present new algorithmic design features. In Section 4, we detailed how the rotating skip list implements a key value store by combining these features. In Section 5, we present the experimental environment, the machines, the benchmarks and the tested data structures. In Section 6, we thoroughly evaluate the performance of non-blocking skip lists. In Section 7, we compare the performance to balanced trees and various synchronisation techniques. In Section 8, we present the related work. Finally, we conclude in Section 9. The optional Appendices A, B and C depict the headers, the interface functions and the maintenance functions of our C code.

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*. While some logarithmic data structures are cache-friendly and others handle high contention effectively, we are not aware of any such structure that combines cache-friendliness with tolerance to contention.

Tolerance to contention. To measure the impact of this problem, let us select the the state-of-the-art skip list from Fraser [13] and the recent no hotspots skip list from Crain et al. [7]. On the one hand, we confirm that Crain's skip list [7]

[1] `http://developers.memsql.com/docs/latest/indexes.html`.

[2] `http://www.arangodb.org/manuals/1.4/dba-manual.pdf`.

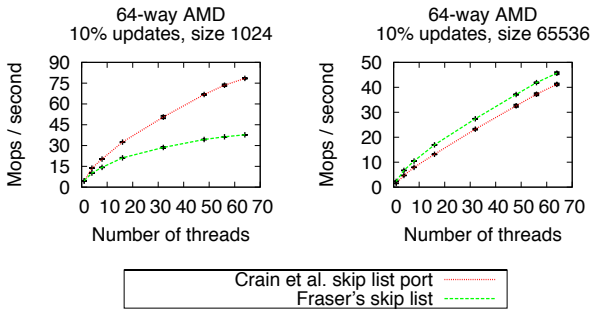


Figure 1. The C implementation of Crain's skip list does not perform well on large dataset while Fraser's skip list does not perform well under contention (update ratio is 10%)

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 [13], 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. 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 that is publicly available [12], to experiment on a 64-core AMD machine (experimental settings are detailed in Section 5).

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 memory locality 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).

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 memory locality, quite high.

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 are in the cache miss rate)

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 [25]. 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. Designing Novel Structures for Multicores

In the context of the logarithmic data structures, multi-threaded accesses translate into having contention hotspots at the top levels of the structure: each modification that involves update on a node of high level in a tree or a tall tower of a skip list is likely to contend with concurrent traversals as the higher a tower or node is, the more likely it will be traversed.

3.1 Avoiding contention hotspots

Decoupling the structure maintenance from regular operations and delegating it to a background thread facilitates restricting contention to the least contended part of the structure: the bottom level.

Periodically, *background threads* traverse the structure and rebalances it if necessary by adjusting the size of towers of a skip list of by rotating a tree. Such restructuring can be achieved lazily, no need to rebalance under contention bursts where performance may be even more affected by the the bottleneck effect than by the asymptotic complexity. This technique lets other threads, say *application threads*, execute update accesses to the data structure efficiently by returning right after marking/inserting a single node in the data structure, hence reducing latency.

The restructuring can also be achieved incrementally by the background threads, no need to block application threads when the restructuring is pending. In a skip list, the maintenance threads could progressively raise towers in regions of a skip list where there are short towers or by rotating a subtree of the tree. The resulting structure may become unbalanced if application threads update the structure faster than the background threads rebalances it. This however only

happens during contention bursts, where multiple threads would anyway contend in a classic data structure.

Moreover one restructuring may compensate the effects of multiple updates. For example, while one insertion on a right subtree may unbalance an AVL tree, the insertion on the left subtree may later on rebalance the tree without rotations. Similarly, removing the entire bottom level of a skip list may simply help shortening all towers at once, without the need to decrease the individual level of each tower.

The decoupling of structural maintenance and regular update operations should be even more efficient when background threads and application threads mutate different regions of the structure. For example, if the former affects only upper levels of an external tree or a skip list whereas the latter only affects its bottom-most level, then contention should be substantially reduced. Though appealing for limiting contention, splitting the structure into regions that can be mutated by distinct threads often adds costly levels of indirections through pointers in the code.

3.2 Maximizing memory locality

Arrays can limit the levels of indirections and are generally more efficient than pointer-based substructures to link the regions of a data structure. This is mainly due to the cache coherence protocol of modern computers that largely exploits memory locality. When fetching a node from memory, the obtained cache line is likely to contain the node that will be accessed next only if consecutive nodes are stored in contiguous locations of the memory. Conversely, a pointer-based linked list representation of nodes does not ensure that consecutive nodes will be part of the same cache line. As memory locality increases cache hits it can reduce the time to access a data by up to two orders of magnitude (the precise cache and memory access latencies are given for AMD Opteron and Intel Xeon mutlicores in Section 5).

A well-known drawback of arrays as opposed to pointer-based structures is the complexity induced by their modification. While inserting a node to or removing a node from a pointer-based linked list simply affects a localised part of the structure, there is no easy way to insert (resp. discard) a location in (resp. from) the middle of an array. As a result, restructuring a data structure makes it difficult to maintain memory locality.

While some skip lists implement tower using arrays [13] to benefit from memory locality, others organize towers as linked lists to simplify modifications. As mentioned in Section 3.1, deleting the bottom level of a skip list can help avoiding contention hotspots, however, it would be too costly to delete the bottom level of several arrays in order to carry out lowering, as this would necessitate the deletion of all the arrays and the creation of new array copies with the lowest level discarded. This leads to a tradeoff between avoiding hotspots and favouring memory locality in existing data structures. This tradeoff motivates the need for novel data structure algorithms.

3.3 Arrays with dynamic content

To combine memory locality with absence of hotspots, one has to use static arrays that contain a dynamically changing sets of consecutive elements. These arrays can speed up data structure traversals by accessing elements in the order they are organised in memory, hence maximising cache hits. A marker element, say ZERO, can be used to indicate the beginning of the array to allow for rapid adjustments of its content upon restructuring. The same modulo arithmetic as the one used to implement a simple rotating buffer can help wrapping elements around the array as indicated in Figure 2.

```
node_t *next = index_array[(ZERO + i)%N];
```

Figure 2. Arrays with dynamic content can easily be accessed through modulo arithmetic with specific marker to limit hotspots and ensure memory locality

Such an array with dynamic content can easily be used to replace each tower of a skip list by a wheel that maintains dynamically the levels of a skip list node. The modulo arithmetic guarantees that increasing indexing past the end of the array wraps around to the beginning of the array. Figure 2 indicates how to access the i^{th} element of the wheel using the wheel capacity N and a global variable ZERO that keeps track of the current lowest level. It returns a pointer to the next node at level i. Incrementing variable ZERO allows to restructure the skip list by lowering all towers in constant time, hence avoiding hotspots.

This approach is different from dynamic arrays or resizable arrays used in programming languages. Dynamic arrays reserve two different spaces for allocations but do not guarantee memory locality across spaces. Resizable arrays allocate contiguous memory blocks, however, they require a re-allocation mechanism often constrained by external fragmentation. Here, the idea is to pre-allocate a sufficiently large block of memory to avoid re-allocation. Note that the size of shared memory in multicore computers tends to rapidly enlarge. Moreover we focus on data structures that have logarithmic time complexity, and such arrays aim to speedup traversal, thus their size could also be kept logarithmic in the size of the structure.

4. The Rotating Skip List

In this section, we present the rotating skip list by describing its implementation of a key-value store. Unlike traditional skip lists, the rotating skip list does not guarantee $O(\log n)$ access time complexity "in expectation". It actually uses a deterministic procedure to guarantee exactly $O(\log n)$ access time complexity when contention disappears. Unlike trees that require a global restructuring to ensure this determinis-

tic guarantee, the rotating skip list limits restructuring, this minimises the contention.

4.1 Key-value store

Key-value stores are important components of data management in widely-distributed systems. They offer the basis for indexes to speed up access to data sets, they can be sharded for scalability, and their simple API has proved popular as a programming model in the NoSQL movement. Our goal is to design a skip list that implements an efficient atomic key-value store abstraction. A key-value store supports at least the following operations:

- put(k,v) - insert the key-value pair $\langle k, v \rangle$ and returns true if the k is absent, otherwise return $\perp$
- delete(k) - remove k and its associated value and return true if k is present in the store, otherwise return false
- get(k) - return the value associated with key k if k is present in the store and return false otherwise

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 be *linearizable* [18]. 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.

4.2 Overview

The rotating skip list employs memory locality to diminish memory traffic and avoids contention hotspots by localizing the modification of update operations (e.g., put and delete in the key-value store scenario) to the least contended part of the data structure. Application threads, whose pseudocode is depicted in Algorithms 1 and 2, may traverse the upper levels but insert and remove elements by simply updating the bottommost level of the structure (lines 56, 61 and 84), where the node information is kept.

Memory locality is achieved by using a rotating array sub-structure, called the *wheel*, instead of the traditional skip list tower, as described in Section 4.5. Freedom from hotspots is achieved by decoupling the insertion/deletion of an element (Alg. 1 and 2) from the structural adaptation of removing/raising the corresponding wheel (Alg. 3). In case of contention peak due to simultaneous update requests, the top of the structure does not suffer concurrent modifications but only the bottom levels are updated. All adjustments to the upper levels are sequentially executed by a dedicated maintenance thread, hence allowing a deterministic adjustment of the levels.

4.3 Structure

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 lower than any other key that can be inserted in the data structure. A *tail* node is used to indicate the end of the data structure, its dummy key is strictly larger than any other possible values. On Figure 3 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, nodes 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 and possibly after reaching the wheel capacity.

The *node* structure contains multiple fields depicted in Algorithm 1 at lines 1–15. It first contains a key-value pair denoted by $\langle k, v \rangle$. The value v may indicate that the node is logically deleted when $v = \perp$ or physically deleted when $v = node$. 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

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[IDX(i, zero)]                 ▷ follow level in wheels
22:     if next = ⊥ ∨ next.k > k then                   ▷ if logically deleted or its 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 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
```

Algorithm 2 The rotating skip list - put and delete functions

```
42: put(k, v):
43:   Find entry to bottom level as lines 17–28
44:   while true do                                   ▷ find the correct node
45:     while node = (val ← node.v) do                ▷ physically removed?
46:       node ← node.prev                            ▷ backtrack
47:     next ← node.next
48:     if next ≠ ⊥ then
49:       next_v ← next.v
50:       if next = next_v then
51:         help_remove(node, next)                   ▷ help remove marked nodes...
52:         continue                                  ▷ ...then retry
53:     if next = ⊥ ∨ next.k > k then                 ▷ position found
54:       if k = node.k then                          ▷ key k is already present
55:         if val = ⊥ then                           ▷ if logically deleted...
56:           if CAS(&node.v, val, v) then return true ▷ ...logically insert
57:         else return false
58:       else                                        ▷ key is not already in the list
59:         new ← new_node(k, v, node, next)          ▷ create new node
60:         if CAS(&node.next, next, new) then        ▷ physically insert
61:           if next ≠ ⊥ then next.prev ← new        ▷ link backward pointer
62:           return true                             ▷ insertion succeeded
63:         else delete_node(new)                     ▷ contention, retry insertion
64:     node ← next                                   ▷ continue the traversal

65: delete(k):
66:   Find entry to bottom level as lines 17–28
67:   while true do                                   ▷ find the correct node
68:     while node = (val ← node.v) do                ▷ physically removed?
69:       node ← node.prev                            ▷ backtrack
70:     next ← node.next
71:     if next ≠ ⊥ then
72:       next_v ← next.v
73:       if next = next_v then
74:         help_remove(node, next)                   ▷ help remove marked nodes
75:         continue                                  ▷ ...then retry
76:     if next = ⊥ ∨ next.k > k then                 ▷ we are done...
77:       if k ≠ node.k then return false
78:       else
79:         if val ≠ ⊥ then
80:           while true do                           ▷ until node is logically deleted
81:             val ← node.v
82:             if val = ⊥ ∨ node = val then          ▷ already deleted or removed
83:               return false
84:             else if CAS(&node.v, val, ⊥) then     ▷ logical delete
85:               if should_help_remove() then        ▷ heuristic to decide
86:                 remove(node.prev, node)           ▷ removal
87:               return true
88:         else return false
89:     node ← next                                   ▷ continue the traversal
```

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). Finally, the *marker* is a special mark used only during physical removal.

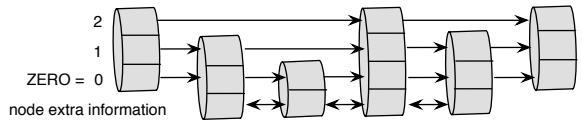
4.4 Traversal

Any traversal, whether it be 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 from line 17 to line 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 0.

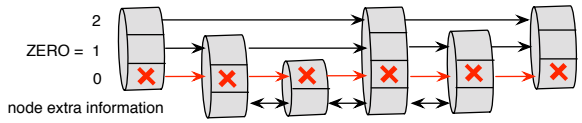
Once the get has reached the bottom level, *node* is actually set to the node with the largest key $k' < k$. If this *node* is physically deleted, the traversal backtracks among deleted nodes at lines 30 and 31 and invokes help_remove to notify the background thread of the nodes to be removed (line 36). Then it updates *next* to the immeditate next node (*node.next*). Note that the traversal can 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)). This requirement imposes to maintain a doubly linked list at the bottom of the structure. Once the get reaches the right position at the bottom level indicated by the fact that $next.k > k$ (line 38), then it checks that the targeted key k is present at some non-deleted node (line 39). If so, it returns the associated value *val*, otherwise it returns $\perp$ 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).

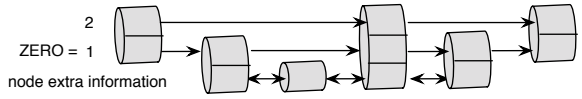
The reason why nodes may be logically deleted is to minimise contention. The delete function (Algorithm 2, lines 65–89) marks a *node* as logically deleted by setting its value to $\perp$. A separate *maintenance thread* is responsible of traversing the bottom level of the skip list to clean up the deleted nodes as described in Section 4.6. 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. It may then decide to make the background thread remove the node based on some heuristic that aims



(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, the bottommost level represents where the node extra information ($\langle k, v \rangle$, *prev*, *next*, *marker*, *level*) is located

to limit contention, before returning true. The same maintenance thread responsible for the clean-up also checks and adjusts the level of the newly inserted nodes. Note that the maintenance thread is unique and is the only thread responsible of updating upper levels, hence proceeding with reduced contention.

4.5 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 as described in Section 3.3. The modulo arithmetic guarantees that increasing the index past the end of an array wraps around to the beginning of the array (see Figure 2). This allows lowering to be done by simply increasing the index of the lowest logical level of all the arrays. 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 116). This causes other threads to stop traversing the index levels before they reach the previous lowest level of pointers (Figure 3 illustrates this graphically). 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 memory locality, 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 that the index *next* references. This extra layer of indirection can hurt performance of the skip list, especially since this indirection is followed by any traversal of the structure.

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)                          ▷ sporadically
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:        lower_skiplist()                         ▷ lower the skip list

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

109: lower_skiplist()_b:
110:   node ← sl.head
111:   zero ← ZERO
112:   while node ≠ ⊥ do                            ▷ up to the tail
113:     node.succs[IDX(0, zero)] ← ⊥                ▷ nullify the wheels
114:     node.level ← node.level − 1                 ▷ decrement this node's level
115:     node ← node.next                            ▷ continue with immediate next node
116:   ZERO ← ZERO + 1                               ▷ remove bottom index level

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

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

4.6 Background thread

The background (or maintenance) thread code is described in Algorithm 3. For the sake of simplicity, we omitted the definition of four functions: raise_node, raise_index_level, is_lowering_necessary and delete_node; the complete code is deferred to Appendix C. It executes a loop (lines 90–99) where it sleeps (line 94). The maintenance thread raises wheels of non-deleted nodes by calling the raise_bottom_level

function (line 95). This raises the level of non-deleted nodes which may make the stucture too high to guarantee logarithmic complexity. In this case, is_lowering_necessary returns true and the lower_skiplist function is called to lower the structure (line 99), progressively lowering all deleted nodes. This periodic adaptation makes it unnecessary to use the traditional pseudo-random generators as the lowering and raising of the towers become deterministic.

The raise_bottom_level (lines 100–108) also cleans up the skip list by removing the logically deleted nodes (line 105). 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 4.7.

The lower_skiplist function (lines 109–116) discards the entire bottom level of the skip list by simply changing the ZERO counter used in the modulo arithmetic, without blocking application threads (as depicted in Figure 3(a)).

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 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 131–134). Note that at the end of the removal (line 136) the *prev* field can be updated without synchronisation as it does not have to be set to the immediate previous node. Whether or not application threads trigger physical deletions themselves is dictated by how unbalanced the skip list is, and this is communicated by the background thread and determined using a heuristic.

4.7 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 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.

4.8 Correctness and progress

Our key-value store implementation is linearizable [18]. 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 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 synchronisation technique for updating and no synchroni-

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

sation 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.

5. Experimental Settings

In this section we describe the multicore machines and the benchmarks used in our experiments as well as the six 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) and another with 4 16-way AMD Opteron 6378 processors (64 ways). 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 MESI 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 [20]. 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 [4].

Data structures and synchronisation 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 synchronisation technique as our rotating skip list, they are non-blocking thanks to the exclusive use of CAS. 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 as our data structure accesses were too long to fit in hardware (the Intel Haswell L1 data cache with hardware lock elision support has a size of 32KiB).

1. Fraser's skip list is the publicly available library [12] developed in C by Fraser. It includes several optimisations compared to the algorithm presented in his thesis [13]. We kept this library unchanged for the sake of fair comparison.
2. Crain's no hotspots skip list is the algorithm presented recently [7]. 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 memory reclamation mechanism. Memory reclamation tends to be costly in our benchmarks.
3. Our lock-based skip list implements the Optimistic Skip List algorithm [16]. It uses the logical deletion technique and allows 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 contention.
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 [10]. Elastic transaction is a relaxed transaction model that was shown particularly promising on pointer-based structures, like skip list.
5. The Speculation-Friendly Binary Search Tree [6] is a binary search tree data structure that exports a key-value store interface and uses transactions for synchronisation. We kept the code from the Sycnchrobench benchmark-suite and used the recommended $\mathscr{E}$-STM software library to synchronise it. This tree algorithm was shown particularly efficient on a travel reservation database application.
6. The Bonsai Tree [5] is a binary balanced tree that exploits the Read-Copy-Update synchronisation technique (RCU) to achieve fast wait-free read-only accesses [14]. Using RCU, updates produce a copy of the data they write to let read-only accesses execute uninterrupted without acquiring locks or accessing any metadata. This data structure was shown particularly effective for storing memory mapping to limit the contention of page faults on the same cache line. As updates are not protected from one another they cannot run concurrently in the original bonsai tree. To allow multiple threads to concurrently update the bonsai tree structure, we added a spinlock that was acquired for the duration of the update access. All read-only

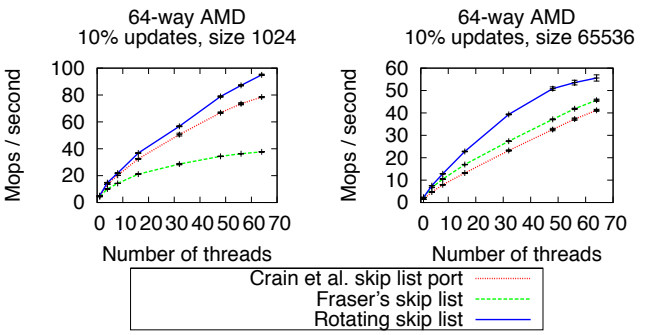


Figure 4. The rotating skip list outperforms both the Crain's and Fraser's skip lists at all thread count under 10% update

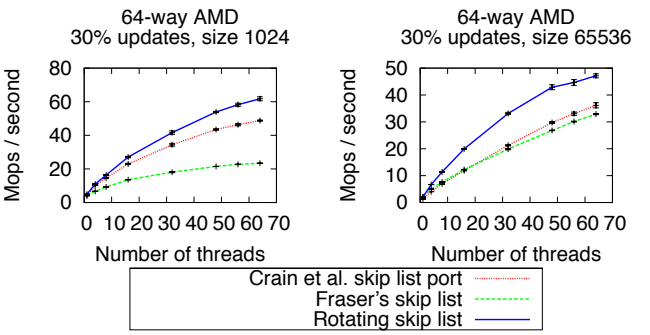


Figure 5. The rotating skip list does not suffer from contention hotspots

accesses were kept unchanged to avoid slowing down the fast RCU-based read-only accesses.

The code of all logarithmic data structures listed above was compiled using gcc v4.7.2 with optimizations (-03) and run on the AMD and the Intel machines listed in Section 5.

Benchmarks. We are currently integrating all 7 logarithmic data structures in the Synchrobench benchmark suite: Synchrobench[3] is a publicly available benchmark suite written in C to evaluate data structures using various synchronisation techniques in a multicore environment. Synchrobench is especially useful to observe performance variations depending on the contention of the system.

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 6 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 7 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 than 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 $\{1,4,8,16,32,48,56,64\}$.

6. 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.

[3] `https://github.com/gramoli/synchrobench`.

6.1 Highest throughput

Figure 4 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 [13], 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 [8] or under simple read-only workloads [1]. As we show in detail below, the higher performance of the rotating skip list stems from the combination of experiencing no hotspots and being cache efficient.

6.2 Tolerance to contention

We increase the update accesses to 30% 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 5 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 structure (2^{10} elements) where the probability of contending increases. Only does the bottom level of the list need synchronisation, which reduces significantly the number of CAS necessary to insert or delete a node without data races. In addition, the higher levels 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.

6.3 Memory locality

We also performed comparative evaluations in settings with no contention at all. Figure 6 indicates the performance of the three skip lists when there are no update operations.

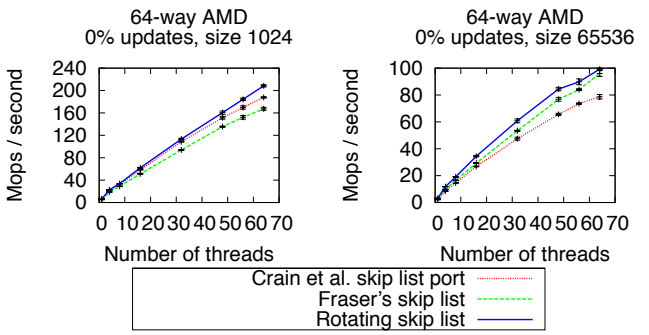


Figure 6. The data locality of the rotating skip list boosts read-only workloads

Fraser's skip list is known to be particularly efficient in such settings [1], 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 synchronise 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. 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.

6.4 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.

6.5 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. This initially populates the structure with (1024) keys that are within a subrange ($[0, 1024]$) of the

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)

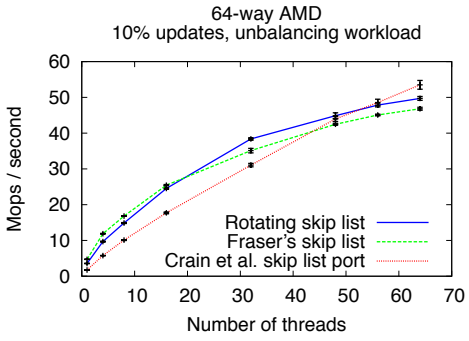


Figure 7. The rotating skip list rebalances effectively under biased workloads

admitted range ($[0, 32768]$); application threads are then inserting/deleting 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 7 depicts the performance results: it is clear that a single maintenance thread in the rotating skip list is enough to raise the new towers inserted by the application threads. In particular, the rotating skip list provides performance superior to Fraser's for all thread counts larger than or equal to 32. 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.

7. Extra Synchronizations and Structures

We also compared the performance of the rotating skip list against two different logarithmic data structures, a binary search tree and another balanced tree, as well as lock-based and transaction-based skip lists.

Synchronization techniques. Our lock-based skip list implements the Optimistic skip list algorithm [16]. 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. (A mentioned previously, due to the limited L1 (data) cache size (32KiB) of the Intel Haswell processor we could not use hardware transactions.) Elastic transaction is a relaxed transaction model that was shown particularly promising on pointer-based structures [10], like skip lists.

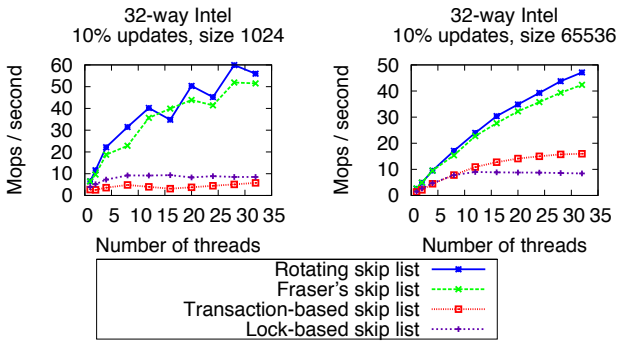


Figure 8. Intel measurements to compare the performance against other synchronisation techniques

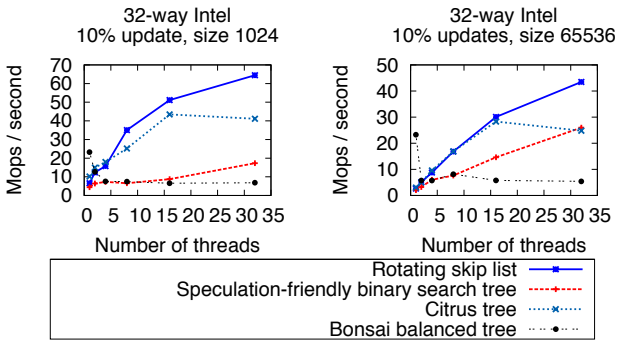


Figure 9. Intel measurements to compare the performance against balanced tree data structures

Figure 8 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.) We ran the same binary of the rotating skip list we tested on the AMD machine in Section 6. An interesting phenomenon is the variation of performance we obtain when changing the number of threads. This behavior is due to the fact that our implementation was not tuned for the cache sizes of the Intel machine that differ from the AMD machine. We intentionally avoided any machine-dependent macro definition to observe the behavior of the exact same implementation on the Intel machine. An interesting result is that even with such a highly varying throughput, the rotating skip list still provide higher peak performance than other data structures.

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 no 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.

We could have tested hardware transactions on an 8-way Haswell processors in the case of 1024 elements, however, recent performance results [24] on a 1024-sized red-black tree with 10% updates indicate a performance peak around 28 M operations/second, which is already 12% slower than the throughput of the rotating skip list on our Intel Xeon machine running only 8 threads.

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 two recent balanced tree data structures shown efficient in database and operating system.

We benchmarked the recent Speculation-Friendly Binary Search Tree [6] that uses optimistic concurrency control and delay the rebalancing under contention peaks to handle the potentially large number of threads of modern multicores. We also benchmarked the recent Bonsai Tree [5], a binary balanced tree that exploits the Read-Copy-Update synchronisation technique (RCU) to achieve very fast wait-free read-only accesses [14].

Figure 9 depicts the performance we obtained when running the rotating skip list, the speculation-friendly tree and the bonsai tree on the Intel machine. Another observation is that the rotating skip list is faster on the Intel machine than on the AMD one at equivalent thread counts (e.g., 32), this is due to the similar cache/memory performance of the two machines and that the clock frequency of the Intel machine is 33% higher than the AMD's (2.1GHz vs. 1.4Ghz). The performance of the Bonsai balanced tree does not scale at all. The main reason is the lock that protects update accesses from each other. While the read-copy-update mechanism is lightweight and allows the Bonsai tree to perform much faster than the two other structures at 1 thread, the lock acquired by all updates prevent performance from increasing at 2 threads even on 10% updates. While read operations do not grab any lock, it looks like the time spent waiting to update is sufficient to dramatically affect the performance. This observation suggests that a better way of exploiting RCU would be to serialize all updates on a single thread and have the remaining $n-1$ threads executing only read-only accesses, however, this would force some of the 16 cores to stay idle.

8. Related Work

Mutual exclusion allows programmers to avoid data races in concurrent programs, however, the related lock metadata management is slow or induces cache traffic which might affect scalability. Some locks, like mutexes are not well-suited as they trigger context switches whose overhead becomes predominant in an in-memory context. Test-and-set spinlocks are prone to the bouncing problem where acquir-

ing a lock invalidates the cache of all threads reading the lock as they are waiting for its release. Other forms of locks, like ticket locks, are detrimental to the performance of the linux kernel [2]. Although some recommend the use of more scalable locks [17], there is a growing interest in non-blocking operations for scalable data structures [21–23].

Various non-blocking data structures were proposed since the mid-1990's to avoid the use of locks and guarantee that the system as a whole always makes progress. Valois [32] was the first to use linked structures to implement a non-blocking dictionary. Harris [15] proposed a technique for decoupling node deletions from physical removals and he used this technique to design a non-blocking linked-list using compare-and-swap (CAS). The same decoupling approach was used by Michael to design a non-blocking linked-list and a list-based non-blocking hash table with advanced memory reclamation [26].

Non-blocking binary search trees and skip lists were proposed to reduce the access complexity. Fraser [13] illustrated a method for constructing a non-blocking binary search tree, although his method requires the use of a multi-word CAS primitive. Ellen et al. [9] designed the first non-blocking binary search trees using single-word CAS for synchronisation, however, it is not balanced. Braginsky and Petrank [3] propose a non-blocking B+-Tree that is implemented using just single-word CAS operations. They report results of better contention handling and higher scalability in comparison to a lock-based B+-Tree. Mao et al. [23]. combined tries with B+-trees to offer lock-free lookups that reduce cache-misses in persistent storage.

Howard and Walpole [19] apply the concept of *relativistic programming* to the problem of designing a concurrent balanced BST, whereby reader and writer threads are exposed to different copies of the data structure so that conflicts are reduced. Their approach offers wait-free reading and is reported to perform well against other implementations, however concurrent transactions are not guaranteed to be linearizable, which the authors argue is an unnecessary condition for concurrent data structures. Crain et al. [6] decouple the rebalancing of a binary search trees from insertions and deletions. They use a transactional approach in order make their tree *speculation-friendly*, which offers another interesting approach to the problem of transforming the balanced BST into a scalable concurrent data structure. As we showed in Section 7, our rotating skip list is substantially more efficient than the speculation-friendly tree.

Skip lists are popular alternative to non-blocking balanced trees due to their simplicity. Fraser has proposed a practical implementation of a non-blocking skip list [13] relying exclusively on CAS for synchronisation and that builds upon the Harris logical deletion technique [15]. His implementation was written in C and includes an epoch based memory reclamation technique. Doug Lea implemented a variant of Fraser's skip list in Java and included it in the JDK java.util.ConcurrentSkipListMap and ConcurrentSkipListSet since version 1.6. This implementation uses a logical deletion technique similar to Fraser's but does not need explicit memory reclamation thanks to the JVM built-in garbage collector. As far as we know Fraser's skip list remains the fastest performing skip list. Recent well-engineered alternatives showed that Fraser's skip list scales very well with the number of cores [8] and achieve performance peak of unequalled throughput of 10^8 operations per second on read-only workloads [1]. Our rotating skip list outperforms Fraser's on all workloads we experimented including read-only ones, and by a multiplying factor of 2.6 achieved under contention.

Sundell and Tsigas [30] have also proposed a non-blocking skip list, which is used to implement a dictionary abstraction. Their approach also uses CAS, test-and-set and fetch-and-add. Fomitchev and Ruppert [11] proposed a non-blocking skip list where lookups physically remove nodes that have been marked for removal, however they do not offer an implementation of their approach. Recently, a shallow skip list was combined with a hash table to store high levels nodes [27]. The resulting skip-trie data structure uses CAS and double-wide double-word-compare-single-swap and has a complexity proportional to the contention and sub-logarithmic in the provisioned key space. All these approaches let threads contend on the upper-levels.

Crain *et al.* recently proposed a skip list [7] that avoids hotspots by delegating the task of maintaining upper-levels to a single thread. It was developed in Java and proposed as an alternative to the state-of-the-art JDK concurrent key-value store algorithm (j.u.c.ConcurrentSkipListMap). Removing hotspots is beneficial as Crain's skip list multiplies the performance of the JDK alternative by 2.5× on 24 cores. Unfortunately, Crain's implementation is heavily tied to the Java programming language. The built-in stop-the-world garbage collector of the JVM prevents the programmer from fully controlling memory reclamation. Implementing similar concepts in a different programming language, like C, requires explicit memory reclamation [25] and hardware-level optimizations [23]. As opposed to our solution, the no hotspots skip list does not target cache-friendliness.

Levandoski et al. [22] proposed a new type of data structure, called a "BW-Tree", which is a cache-friendly version of a B+-Tree implemented using CAS to reduce blocking (the structure only blocks when it needs to fetch pages from stable storage). The B+-Tree is optimised to work with flash-based stable storage, and is organised around a mapping table that virtualises the location and size of pages. Updates are done by prepending update deltas to the prior page state, which reduces the number of cache invalidations. 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.

9. Conclusion

Data structures whose accesses have logarithmic time complexity naturally distribute concurrent threads of execution to different paths from a single entry point (e.g., the root of the tree) to different nodes. These data structures suffer from update operations modifying the nodes that are the most frequently accessed hence producing contention hotspots.

We proposed new algorithmic design choices to implement efficient data structures for multicore. We illustrated these choices by devising a novel data structure, the rotating skip list, which is, as far as we know, the most efficient skip list to date.

We draw four main conclusions out of this work: (1) The layout of nodes in a skip list is a good alternative to the hierarchical layout of a balanced tree as restructuring a skip list does not need to update the single entry point, hence limiting contention. (2) Static arrays whose content is adjustable through modulo arithmetic are instrumental in improving performance by increasing memory locality and cache hits on modern multicore machines. (3) The exclusive use of hardware primitives, like CAS, to synchronise data structures avoids the overhead of software transactions or the cache traffic of some locks on multicore machines. (4) The rotating skip list achieves peak performance of 2×10^8 operations/second on commodity multicore machines indicating that it could be the structure of choice to implement key-value stores.

Future work includes the design of other multicore data structures and the modification of our rotating skip list to support an extended key-value store interface including range queries and get_max.

References

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

[2] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium, Ottawa, Canada*. Citeseer, 2012.

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

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

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

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

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

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

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

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

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

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

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

[14] 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.

[15] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, pages 300–314. Springer, 2001.

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

[17] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kauffmann, 2008.

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

[19] P. Howard and J. Walpole. Relativistic red-black trees. Technical report, Portland State University, 2010.

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

[21] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, pages 73–84. ACM, 2013.

[22] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, 2013.

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

[24] A. Matveev and N. Shavit. Reduced hardware norec: An opaque obstruction-free and privatizing hytm. In *TRANSACT*, 2014.

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

[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] N. Shavit. Data structures in the multi-core age. *Commun. ACM*, 2011.

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

[31] H. Sutter. Choose concurrency-friendly data structures. *Dr. Dobb's Journal*, June 2008.

[32] J. D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, 1996.

A. Header definitions

```
#include <atomic_ops.h>

#define VOLATILE /* volatile */
#define BARRIER() asm volatile("" ::: "memory")

#define CAS(_m, _o, _n) AO_compare_and_swap_full(\
((VOLATILE AO_t*) _m), ((AO_t) _o), ((AO_t) _n))

#define FAI(a) AO_fetch_and_add_full((VOLATILE AO_t*) (a), 1)
#define FAD(a) AO_fetch_and_add_full((VOLATILE AO_t*) (a), -1)

/*
 * Allow us to efficiently align and pad structures so that
 * shared fields do not cause contention on thread-local
 * or read-only fields.
 */
#define CACHE_PAD(_n) char __pad ## _n [CACHE_LINE_SIZE]
#define ALIGNED_ALLOC(_s)                                   \
    ((void *)(((unsigned long)malloc((_s)+CACHE_LINE_SIZE*2) +  \
        CACHE_LINE_SIZE - 1) & ~(CACHE_LINE_SIZE-1)))

#define CACHE_LINE_SIZE 64

#define IDX(_i, _z) ((_z) + (_i)) % MAX_LEVELS

unsigned long sl_zero;

/* bottom-level nodes */
typedef VOLATILE struct sl_node node_t;
struct sl_node {
        unsigned long   level;
        struct sl_node  *prev;
        struct sl_node  *next;
        unsigned long   key;
        void            *val;
        struct sl_node  *succs[MAX_LEVELS];
        unsigned long   marker;
        unsigned long   raise_or_remove;
};

/* the skip list set */
typedef struct sl_set set_t;
struct sl_set {
        struct sl_node  *head;
};

typedef VOLATILE struct sl_node node_t;
struct sl_node {
        unsigned long   level;
        struct sl_node  *prev;
        struct sl_node  *next;
        unsigned long   key;
        void            *val;
        struct sl_node  *succs[MAX_LEVELS];
        unsigned long   marker;
        unsigned long   raise_or_remove;
};
```

B. Interface functions

```
static int sl_finish_contains(unsigned int key,
                              node_t *node,
                              void *node_val, ptst_t *ptst)
{
  int result = 0;

  assert(NULL != node);

  if ((key == node->key) && (NULL != node_val)) {
    result = 1;
  }

  return result;
}


static int sl_finish_delete(unsigned int key, node_t *node,
                            void *node_val, ptst_t *ptst)
{
  int result = -1;

  assert(NULL != node);

  if (node->key != key)
    result = 0;
  else {
    if (NULL != node_val) {
      /* loop until we or someone else deletes */
      while (1) {
        node_val = node->val;
        if (NULL == node_val || node == node_val) {
          result = 0;
          break;
        }
        else if (CAS(&node->val, node_val, NULL)) {
          result = 1;
          if (bg_should_delete) {
            if (CAS(&node->raise_or_remove, 0, 1)) {
              bg_remove(node->prev, node,
                        ptst);
            }
          }
          break;
        }
      }
    } else {
      /* Already logically deleted */
      result = 0;
    }
  }

  return result;
}


static int sl_finish_insert(unsigned int key, void *val,
                            node_t *node, void *node_val,
                            node_t *next, ptst_t *ptst)
{
  int result = -1;
  struct sl_node *new, *temp;

  if (node->key == key) {
    if (NULL == node_val) {
      if (CAS(&node->val, node_val, val))
        result = 1;
    } else {
      result = 0;
    }
  } else {
    new = node_new(key, val, node, next, 0, ptst);
    if (CAS(&node->next, next, new)) {
      if (NULL != next) {
          next->prev = temp;
      }
      result = 1;
    } else {
      node_delete(new, ptst);
    }
  }

  return result;
}

/*
 * the get function has the same code as the contains
 * but returns the value associated with a key, not 1.
 */
int sl_do_operation(set_t *set, sl_optype_t optype,
```

```
                        unsigned int key, void *val)
{
  node_t *item = NULL, *next_item = NULL;
  node_t *node = NULL, *next = NULL;
  node_t *head = set->head;
  void *node_val = NULL, *next_val = NULL;
  int result = 0;
  ptst_t *ptst;
  unsigned long zero, i;

  assert(NULL != set);

  ptst = ptst_critical_enter();

  zero = sl_zero;
  i = set->head->level - 1;

  /* find an entry-point to the node-level */
  item = head;
  while (1) {
    next_item = item->succs[IDX(i,zero)];

    if (NULL == next_item || next_item->key > key) {

      next_item = item;
      if (zero == i) {
        node = item;
        break;
      } else {
        --i;
      }
    }
    item = next_item;
  }

  /* find the correct node and next */
  while (1) {
    while (node == (node_val = node->val)) {
      node = node->prev;
    }
    next = node->next;
    if (NULL != next) {
      next_val = next->val;
      if (next_val == next) {
        bg_help_remove(node, next, ptst);
        continue;
      }
    }
    if (NULL == next || next->key > key) {
      if (CONTAINS == optype)
        result = sl_finish_contains(key, node,
                                    node_val,
                                    ptst);
      else if (DELETE == optype)
        result = sl_finish_delete(key, node,
                                  node_val,
                                  ptst);
      else if (INSERT == optype)
        result = sl_finish_insert(key, val, node,
                                  node_val, next,
                                  ptst);
      if (-1 != result)
        break;
      continue;
    }
    node = next;
  }

  ptst_critical_exit(ptst);

  return result;
}
```

C. Background functions

```
/**
 * bg_loop - loop for maintaining index levels
 * @args: void* args as per pthread_create requirements
 *
 * Returns a void* value as per pthread_create requirements.
 * Note: Do this loop forever while the program is running.
 */
static void* bg_loop(void *args)
{
  node_t  *head  = set->head;
  int raised = 0; /* keep track of if we raised index level */
  int threshold;  /* for testing if we should lower index level */
  unsigned long i;
  ptst_t *ptst = NULL;
  unsigned long zero;

  assert(NULL != set);
  bg_counter = 0;
  bg_go = 0;
  bg_should_delete = 1;
  BARRIER();

  while (1) {
    usleep(bg_sleep_time);
    if (bg_finished)
      break;
    zero = sl_zero;

    bg_non_deleted = 0;
    bg_deleted = 0;
    bg_tall_deleted = 0;

    // traverse the node level and try deletes/raises
    raised = bg_trav_nodes(ptst);

    if (raised && (1 == head->level)) {
      // add a new index level

      // nullify BEFORE we increase the level
      head->succs[IDX(head->level, zero)] = NULL;
      BARRIER();
      ++head->level;
    }

    // raise the index level nodes
    for (i = 0; (i+1) < set->head->level; i++) {
      assert(i < MAX_LEVELS);
      raised = bg_raise_ilevel(i + 1, ptst);

      if ((((i+1) == (head->level-1)) && raised)
          && head->level < MAX_LEVELS) {
        // add a new index level

        // nullify BEFORE we increase the level
        head->succs[IDX(head->level,zero)] = NULL;
        BARRIER();
        ++head->level;
      }
    }

    // if needed, remove the lowest index level
    threshold = bg_non_deleted * 10;
    if (bg_tall_deleted > threshold) {
      if (head->level > 1) {
        bg_lower_ilevel(ptst);
      }
    }

    if (bg_deleted > bg_non_deleted * 3) {
      bg_should_delete = 1;
      bg_stats.should_delete += 1;
    }
    else {
      bg_should_delete = 0;
    }
    BARRIER();
  }

  return NULL;
}

/**
 * bg_lower_ilevel - lower the index level
 */
void bg_lower_ilevel(ptst_t *ptst)
{
  unsigned long zero = sl_zero;
```

```
  node_t  *node = set->head;
  node_t  *node_next = node;

  ptst = ptst_critical_enter();

  if (node->level-2 <= sl_zero)
    return; /* no more room to lower */

  /* decrement the level of all nodes */

  while (node) {
    node_next = node->succs[IDX(0,zero)];
    if (!node->marker) {
      if (node->level > 0) {
        if (1 == node->level && node->raise_or_remove)
          node->raise_or_remove = 0;
        /* null out the ptr for level being removed */
        node->succs[IDX(0,zero)] = NULL;
        --node->level;
      }
    }
    node = node_next;
  }

  /* remove the lowest index level */
  BARRIER(); /* do all of the above first */
  ++sl_zero;

  ptst_critical_exit(ptst);
}

/**
 * bg_help_remove - finish physically removing a node
 * @prev: the node before the one to remove
 * @node: the node to finish removing
 * @ptst: per-thread state
 *
 * Note: This operation will only be carried out if @node
 * has been successfully marked for deletion (i.e. its value
 * points to itself, and the node must now be deleted). First
 * we insert a marker node directly after @node. Then, if
 * no nodes have been inserted in between @prev and
 * @node, physically remove @node and the marker
 * by pointing @prev past these nodes.
 */
void bg_help_remove(node_t *prev, node_t *node, ptst_t *ptst)
{
  node_t *n, *new, *prev_next;
  int retval;

  assert(NULL != prev);
  assert(NULL != node);

  if (node->val != node || node->marker)
    return;

  n = node->next;
  while (NULL == n || !n->marker) {
    new = marker_new(node, n, ptst);
    CAS(&node->next, n, new);

    assert (node->next != node);

    n = node->next;
  }

  if (prev->next != node || prev->marker)
    return;

  /* remove the nodes */
  retval = CAS(&prev->next, node, n->next);
  assert (prev->next != prev);

  if (retval) {

    node_delete(node, ptst);
    marker_delete(n, ptst);
  }

  /*
   * update the prev pointer - we do not need synchronisation here
   * since the prev pointer does not need to be exact
   */
  prev_next = prev->next;
  if (NULL != prev_next)
    prev_next->prev = prev;
}

/**
 * bg_remove - start the physical removal of @node
 * @prev: the node before the one to remove
 * @node: the node to remove
 *
 * Note: we only remove nodes that are of height 1 (i.e. they
 * do not have index nodes above). Nodes with index items are
 * removed a different way, using index height changes.
 */
void bg_remove(node_t *prev, node_t *node, ptst_t *ptst)
{
  assert(NULL != node);

  if (0 == node->level) {
    /* only remove short nodes */
    CAS(&node->val, NULL, node);
    if (node->val == node)
      bg_help_remove(prev, node, ptst);
  }
}
```