No Hot Spot Non-Blocking Skip List

Tyler Crain
IRISA
Rennes, France
tyler.crain@irisa.fr

Vincent Gramoli
NICTA and University of Sydney
Sydney, Australia
vincent.gramoli@sydney.edu.au

Michel Raynal
IRISA, Institut Universitaire de France
Rennes, France
raynal@irisa.fr

***Abstract*—This paper presents a new non-blocking skip list algorithm. The algorithm alleviates contention by localizing synchronization at the least contended part of the structure without altering consistency of the implemented abstraction.**

The key idea lies in decoupling a modification to the structure into two stages: an eager abstract modification that returns quickly and whose update affects only the bottom of the structure, and a lazy selective adaptation updating potentially the entire structure but executed continuously in the background.

On SPECjbb as well as on micro-benchmarks, we compared the performance of our new non-blocking skip list against the performance of the JDK non-blocking skip list. The results indicate that our implementation can me more than twice as fast as the JDK skip list.

***Index Terms*—contention; data structure; lock-freedom**

I. Introduction

Skip lists are increasingly popular alternatives to B-trees in main-memory databases like *memsql* as their elements are traversed in order and without the need of latches. In short, a skip list is a linked structure that diminishes the linear asymptotic complexity of a linked list by having *index-items* on top of nodes that together form *towers* with additional shortcuts pointing towards other towers located further in the list [15]. These shortcuts allow operations to complete in $O(\log n)$ steps in expectation by letting traversals skip lower nodes through higher level shortcuts. The drawback of employing shortcuts is however to require additional maintenance at multiple levels each time some data is stored or discarded. This increases the probability of multiple *threads* (or processes) interfering on the same shared data.

Maintaining the skip list in the case of concurrent traversals creates contention *hot spots* typically located at the top of the skip list towers. More specifically, the higher a tower is, the more likely it will be accessed and the more contention its update will incur. These hot spots become important bottlenecks on modern machines with a large amount of cores, which typically translates into significant performance losses.

In the light of the impact of contention on performance, we propose a new *Contention-Friendly (CF)* non-blocking skip list algorithm experiencing contention only at the bottom level that consists of the *node list*, thus avoiding the contention hot spots other concurrent skip lists suffer from. Even though this new concurrent skip list alleviates the contention of modern multi-cores, it does not relax the correctness of the abstraction it implements. To accomplish this our skip list benefits from a genuine decoupling of each update access into an eager abstract modification and a selective lazy structural adaptation:

- Concurrent *eager abstract modifications* consist in modifying the abstraction while minimizing the impact on the structure itself and returning as soon as possible for the sake of responsiveness.
- The *lazy selective adaptation*, which executes in the background, adapts the skip list structure based on abstract changes by re-arranging elements or garbage collecting and physically removing logically deleted ones.

When applying this decoupling to the map or set abstraction, it translates into: (1) splitting an element *insertion* into (a) the insertion phase at the bottom level of the skip list and (b) the structural adaptation responsible for updating pointers at its higher levels, and (2) splitting an element *removal* into (a) a logical deletion marking phase and (b) its physical removal and garbage collection. Hence, the decoupling allows multiple threads to update the bottom-most level (i.e. the node list) of the skip list while a single thread keeps adapting higher levels in the background.

The consequence of our decoupling is twofold. First, it shortens operations in order for them to return just after the abstract access, hence diminishing their latency. Second, it postpones the structural adaptation to selected parts of the structure in order to avoid temporary load bursts and hot spots, hence diminishing contention.

The decoupling also impacts the implementation design in three main ways. First, decoupling the structural adaptations from responsive abstract operations lets us adapt deterministically the heights of the towers to obtain a more balanced structure than with classic tightly-coupled and pseudo-random modifications. Second, in the case where a large amount of short towers have been removed (leaving too many tall towers), we decrease the heights of all towers by removing the bottom-most index-item list of the structure rather than modifying the frequently accessed higher levels. Third, this decoupling allows us to centralize all adaptation tasks on a single thread, meaning that only accesses to the node list level need to be synchronized, hence further reducing latency.

Additionally, our CF skip list is *non-blocking*, ensuring that the system as a whole always makes progress. In particular, no threads ever block waiting for some resources. This property guarantees that one slow thread does not affect the overall system performance, representing an appealing feature for modern heterogeneous machines that tend to embed processors or cores of distinct clock frequencies. Finally, the non-blocking property makes our skip list fault tolerant as if one thread crashes, it neither makes the skip list inconsistent nor does it

1063-6927/13 $26.00 © 2013 IEEE
DOI 10.1109/ICDCS.2013.42

IEEE computer society

prevent other threads from accessing resources. In particular, if the adaptation thread slows down or crashes then system performance may be affected but safety and progress are preserved.

Evaluation on a micro-benchmark and on the SPECjbb main-memory database benchmark [16] is given. More precisely, we compare our algorithm's performance against the Java adaptation by Doug Lea of Harris, Michael and Fraser's algorithms [9], [13], [8]. This implementation is probably one of the most widely used non-blocking skip lists today and is distributed within the Java Development Kit (JDK). The results observed on a 24-core AMD machine shows a speedup factor of up to 2.5. Therefore, our work answers by the affirmative the open question "is compromising randomness under contention effective?"[1]

Section II describes the related work. Section III depicts how to make a skip list contention-friendly. Section IV describes in detail our CF non-blocking skip list algorithm. Section V presents the experimental results. Section VI presents extensions applicable to our algorithm and Section VII concludes. Please refer to the companion technical report [2] for the proofs of correctness and progress of our algorithm.

II. Related Work

Decoupling tree updates into multiple tasks has proved beneficial for memory management [5] and efficiency [14], [3]. In particular, the work in this paper is motivated by our previous decoupling of memory transactions updating a binary search tree [3]. As far as we know, this idea has never been applied to skip lists before.

The deletion of an element in various data structures has been decoupled to avoid blocking. Tim Harris proposed to mark elements for deletion using a single bit prior to physically removing them [9]. This bit corresponds typically to the low-order bit of the element reference that would be unused on most modern architectures. The decoupling into a logical deletion and a physical removal allowed Harris to propose a non-blocking linked list using exclusively compare-and-swap (CAS) for synchronization. The same technique was used by Maged Michael to derive a non-blocking linked list and a non-blocking hash table with advanced memory management [13] and by Keir Fraser to develop a non-blocking skip list [8].

Doug Lea exploited these techniques to propose a non-blocking skip list implementation in the JDK [12]. For the sake of portability, an element is logically deleted by nullifying a value reference instead of incrementing a low-order bit. The resulting algorithm is quite complex and implements a *map*, or dictionary, abstraction. A tower of height ℓ comprises $\ell - 1$ *index-items*, one above the other with each being part of an *index-item* list level, under which at the bottom level a *node* is used as part of the *node list* to store the appropriate $\langle key, value\rangle$ pair of the corresponding element. Our implementation uses the same null marker for logical deletion, and we employ the same terminology to describe our algorithm.

Sundell and Tsigas built upon the seminal idea by Valois [18] of constructing non-blocking dictionaries using linked structures [17]. They propose to complement Valois' thesis by specifying a practical non-blocking skip list that implements a dictionary abstraction. The algorithm exploits the logical deletion technique proposed by Harris and uses three standard synchronization primitives that are test-and-set, fetch-and-add and CAS. The performance of their implementation is shown empirically to scale well with the number of threads on an SGI MIPS machine. The logical deletion process that is used here requires that further operations help by marking the various levels of a tower upon discovering that the bottommost node is marked for deletion. Further helping operations may be necessary to physically remove the tower.

Fomitchev and Ruppert [7] proposed a non-blocking skip list algorithm whose lookups physically remove nodes they encounter to avoid traversing superfluous towers. This approach differs from the former one where a superfluous tower is traversed by the lookup, while marking all its nodes as deleted. Fomitchev and Ruppert also use the logical deletion mechanism for tower removal by first having its bottommost node marked for removal, then its topmost one. Other operations help removing a tower in an original way by always removing a logically deleted tower to avoid further operations to unnecessarily backtrack. We are unaware of any existing implementation of this algorithm.

In contrast with these three skip lists algorithms, our approach is to decouple the abstract modification from the selective structural adaptation, avoiding contention hot spots by having synchronisation occurring only at the bottom level of the structure.

Finally, transactional memories can be used to implement concurrency skip lists, however, they may restrict their concurrency [8] or use implicit locks [6].

III. Towards Contention-Friendliness

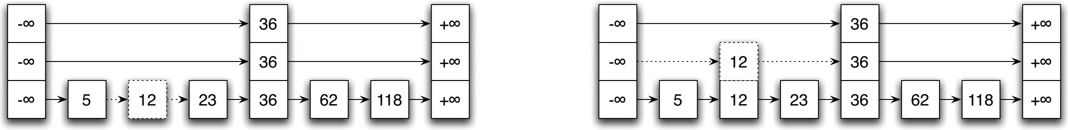
In this section, we give an overview of the technique to make the skip list contention-friendly. Our CF skip list aims at implementing a correct *map*, or dictionary, abstraction as it represents a common key-value store example. The correctness criterion ensured here is linearizability [11].

For the sake of simplicity our map supports only three operations: (i) insert adds a given key-value pair to the map and returns true if the key is not already present, otherwise it returns false; (ii) delete removes a given key and its associated value from the map and returns true if the key was present, otherwise it returns false; (iii) contains checks whether a given key is present and returns true if so, false otherwise. Note that these operations correspond to the putIfAbsent, remove, and containsKey method of the java.util.concurrent.ConcurrentSkipListMap.

A. Eager abstract modification

Previous skip lists generally maintain a tower height distribution so that the probability of a tower i to have height ℓ is $\Pr[height_i = \ell] = 2^{-O(\ell)}$, hence as part of each updating

[1] This question was raised in the extended version of the optimistic skip list [10] entitled "A Provably Correct Scalable Concurrent Skip List".



(a) Inserting horizontally in the skip list

(b) Adapting vertically the skip list structure

Fig. 1. Decoupling the eager abstract insertion from the lazy selective adaptation

abstract operation the invariant is checked and the structure is accordingly updated. Even though an update to the abstraction may only need to modify a single location to become visible, its associated structural adaptation is a global modification that could potentially conflict with any concurrent update.

In order to avoid these additional conflicts, when a node is inserted in the CF skip list only the node list level is modified and the additional structural modification is postponed until later. This decoupling prevents an insertion from updating up to $O(\log n)$ levels, thus reducing contention and making the update cost of each operation more predictable.

As an example, consider the insertion of an element with key 12. Our insertion consists in updating only the node list level of the structure by adding a new node of height 1, leading to Figure 1(a) where dashed arrows indicate the freshly modified pointers. The key 12 now exists in the set and is visible to future operations, but the process of raising this same tower by linking at index-item levels is deferred until later.

It is noteworthy that executing multiple abstract modifications without adapting the structure does no longer guarantee the logarithmic step complexity of the accesses. Yet this happens only under contention, precisely when this asymptotic complexity may not be the predominant factor of performance.

B. Lazy selective structural adaptation

In order to guarantee the logarithmic complexity of accesses when there is no contention the structure needs to be adapted by setting the next pointers at upper (index-item) levels as soon as possible while avoiding an increase in contention. Figure 1(b) depicts the lazy structural adaptation corresponding to the insertion of tower 12: the insertion at a higher (index-item) level of the skip list is executed as a structural adaptation (separated from the insertion), which produces eventually a good distribution of tower heights.

1) Laziness to avoid contention bursts: The structural adaptation is *lazy* because it is decoupled from the abstract modifications and is executed by an independent thread. (A multi-threaded structural adaptation is discussed in Section VI-B.) Hence many concurrent abstract modifications may have accessed the skip list while no adaptations have completed yet. We say that the decoupling is *postponed* from the system point of view.

This postponement has several advantages. An important advantage is to enable merging of multiple adaptations in one simplified step: only one traversal is sufficient to adapt the structure after a burst of abstract modifications. Another interesting aspect is that it gives a chance for insertions to execute faster: Not only does an insert return without modifying the index-item levels, if the element to be inserted is already in the list, but is marked as logically deleted, then the insertion simply needs to logically insert the element by unmarking it. This avoids the insertion from having to allocate a new node and modify the structure.

2) Selectivity to avoid contention hot-spots: The abstract modification of a delete simply consists of marking the node as logically deleted without modifying the actual structure. Importantly, the subsequent structural adaptation selects for physical removal the nodes whose physical removal would induce the least contention. More precisely, only the nodes without towers (i.e. nodes that are not linked to index-item lists) are removed by this operation.

For example, the removal of a tall tower, say the one with value 36 in Figure 1(b), would typically induce more contention than the removal of a node with a shorter tower, say the one with value 62 spanning just the node list level. The reason is twofold. First removing a tower spanning ℓ levels boils down to updating $O(\ell)$ pointers, hence removing the element with value 36 requires updating at least 3 pointers while the element with value 62 requires updating 1 pointer. Second, the organization of the skip list implies that the higher level pointers are traversed more frequently, hence the removal of tower 36 typically conflicts with every operation concurrently traversing this structure whereas the next pointer of element 62 is unlikely to be accessed by as many concurrent traversals.

IV. The Non-Blocking Skip List

In this section, we present our CF non-blocking skip list. Algorithm 1 depicts the algorithm of the eager abstract operations while Algorithm 2 depicts the algorithm of the lazy selective adaptation. CAS operations are used for synchronization. The bottom level (the node list) of the skip list is made up of a doubly linked list of nodes as opposed to the Java ConcurrentSkipListMap, which uses a singly linked list. Each node has a *prev* and *next* pointer, a key *k*, a value *v*, an integer *level* indicating the number of levels of lists this node is linked into, a *marker* flag indicating whether the node is a marker node (used during physical removals in order to prevent lost insert scenarios).

As depicted in Figure 2, the upper levels are made up of singly linked lists of items called *index-items*. Each of these items has a *next* pointer that points to the next item in the linked list, a *down* pointer that points to the linked list of

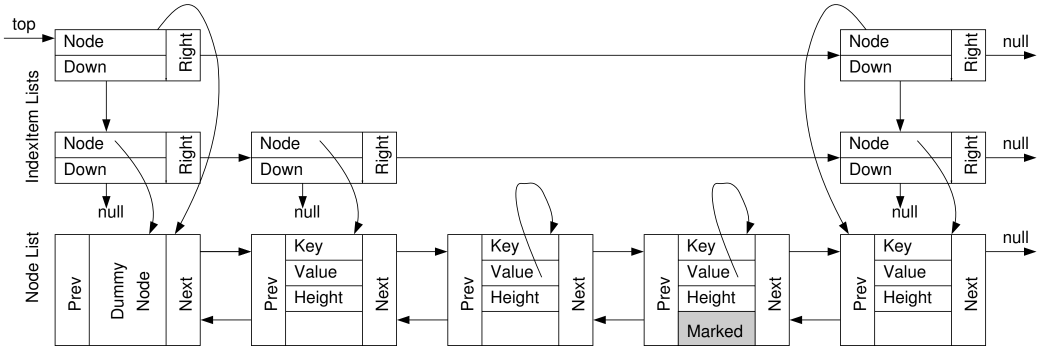


Fig. 2. The contention-friendly non blocking skip list structure

index-items one level below (the bottom level of index-items have $\perp$ for their *down* pointers), and a *node* pointer that points to the corresponding node in the doubly linked list. The list is initialized with a single dummy node with a tower of the maximum starting height. The pointer *top* always points to the first element of the highest index-item list, all traversals start from this pointer.

We use the logical deletion technique [9] by nullifying the v field used to hold the value of the node. If $v = \perp$, then we say that the node is logically deleted. After logical deletion, in order to indicate that a node has been (or is in the process of being) physically removed from the list, the v field is set to point to the node itself (for example a node n has been or is being physically removed if $n.v = n$). This technique is used to ensure the safety of the non-blocking protocol, the details of which are discussed in Section IV-B. In Figure 2 the third node is in the process of being removed and the fourth node is a marker.

A. Abstract operations

Algorithm 1 depicts operations contains, insert and delete. These operations traverse the skip list by executing the same procedure do_operation until line 54 where they execute a specific procedure finish(contains, $*$), finish(delete, $*$) or finish(insert, $*$). Due to concurrent modifications to the skip-list, these procedures might not be able to complete their abstract operation at the current node. In this case $\perp$ is returned to do_operation and the traversal continues (line 57), otherwise true or false is returned and the operation is completed.

1) The traversal: The skip list traversal classically executes from left to right and from top to bottom. More precisely, the do_operation procedure starts by executing the while loop (lines 33-43) which traverses the towers starting from the highest level pointed to by the *top* pointer. Each index-item list is traversed by following the *right* pointers of index-items moving towards increasingly larger keys. A traversal of a level stops either if it reaches the end of the list at the current level or if it reaches a tower linking to a node with a key larger than k (line 35). At this point the traversal continues on the index-item list level below by following the *down* pointer of the current index-item (line 36). If there are no index-item lists below then the while loop is exited (line 39) and the traversal continues on the list of nodes. The index-item list traversal stops immediately if a node with key k is found (lines 40-42).

The traversal then continues on the doubly linked list of nodes (cf. the while loop on lines 44-58) by following the current node's *next* pointer. However, the traversal may reach a node that has been concurrently physically removed ($node.v = node$), in which case it backtracks by following the node's *prev* pointer until it reaches a node guaranteed to be in the list ($node.v \neq node$) this ensures the traversal does not miss newly inserted nodes. In order to ensure non-blocking progress the traversal may also help with the physical removal of the next node if it detects that $next.v = next$ (line 50).

2) The finish: Once the traversal has found the end of the list or a node with key larger than k then the appropriate finish is executed for the current operation type. The finish(contains, $*$) simply checks whether *node* has key k (line 62). A node with value $\perp$ indicates a logically deleted node in which case it is known that k is not in the set. finish(contains, $*$) returns true if key k is found at a non-deleted node.

The finish(delete, $*$) looks for a non-logically deleted node ($n.v \neq \perp$) (line 71) with key k (line 68). If such a node is not found false is returned, otherwise the node is marked as deleted using CAS (line 72). If the CAS fails, the traversal continues as this indicates a concurrent modification. If the CAS succeeds, remove is called to physically remove the node (line 73). This operation will only physically remove nodes of height 1 (i.e. nodes not linked to any index-items lists), other physical removals are dealt with by the adapting thread, the details of this operation is described in section IV-B.

The finish(insert, $*$) operation starts by checking if a node with key k is already in the list (line 79). An interesting implication of separating the structural adaptation is the ability to have lighter insertions. An insert is executed "logically" if it encounters a node with key k that is marked as logically deleted (lines 80) by unmarking it using a CAS (line 81), returning true on success. Otherwise if the node has not been marked as deleted then false is returned (line 83). If no node with key k is found then a new node is allocated (line 85), has its pointers and values set (lines 21-23) and is added to the list using a CAS. If the CAS succeeds the *prev* pointer of the *next* node is set to point to the new node (note that synchronization

Algorithm 1 Contention-friendly non-blocking skip list – abstract operations by process p

```
1:  State of node:
2:     node a record with fields:
3:        $k \in \mathbb{N}$, the node key
4:        $v$, the node's value, a value of $\perp$ indicates
5:           the node is logically deleted
6:        marker $\in \{true, false\}$, indicates if this is
7:           a marker node
8:        next, pointer to the next node in the list
9:        prev, pointer to the previous node in the list
10:       level, integer indicating the level of the node,
11:          initialized to 0

12: State of index-item:
13:    item a record with fields:
14:       right, pointer to the next
15:          item in the SkipList
16:       down, pointer to the IndexItem
17:          one level below in the SkipList
18:       node, pointer a node in the list
19:          at the bottom of the SkipList

20: setup-node$(node, next, k, v)_p$:
21:    $new.k \leftarrow k$; $new.v \leftarrow v$
22:    $new.prev \leftarrow node$
23:    $new.next \leftarrow next$
24:    return new

25: contains$(k)_p$:
26:    return do_operation(contains, $k$, $*$)

27: delete$(k)_p$:
28:    return do_operation(delete, $k$, $*$)

29: insert$(k, v)_p$:
30:    return do_operation(insert, $k$, $v$)

31: do_operation$(op\text{-}type, k, v)_p$:
32:    $item \leftarrow top$                          ▷ start traversing from the top
33:    while true do
34:       $next\_item \leftarrow item.right$          ▷ traverse the list
35:       if $next\_item = \perp \lor next\_item.node.k > k$ then
36:          $next\_item \leftarrow item.down$        ▷ go down a level
37:          if $next\_item = \perp$ then             ▷ bottom level reached
38:             $node \leftarrow item.node$
39:             break()
40:       else if $next\_item.node.k = k$ then
41:          $node \leftarrow item.node$              ▷ found the node
42:          break()
43:       $item \leftarrow next\_item$
44:    while true do                                  ▷ while undone
45:       while $node = (val \leftarrow node.v)$ do
46:          $node \leftarrow node.prev$              ▷ go to still present nodes
47:       $next \leftarrow node.next$                 ▷ load the next node
48:       $next\_val \leftarrow next.v$
49:       if $(next \neq \perp \land next\_val = next)$ then
50:          help-remove$(node, next)$
51:          continue()
52:       if $(next = \perp \lor next.k > k)$ then
53:          $result \leftarrow$                      ▷ finish the contains/delete/insert
54:             finish$(op\text{-}type, k, v, node, val, next, next\_val)$
55:          if $result \neq \perp$ then
56:             break()                               ▷ done!
57:          continue()                               ▷ cannot finish due to concurrency
58:       $node \leftarrow next$                      ▷ continue traversal
59:    return result

60: finish$(contains, k, v, node, val, next, next\_val)_p$:
61:    $result \leftarrow$ false
62:    if $node.k = k$ then
63:       if $(v \neq \perp)$ then                    ▷ check for logical delete
64:          $result \leftarrow$ true
65:    return result

66: finish$(delete, k, v, node, val, next, next\_val)_p$:
67:    $result \leftarrow \perp$
68:    if $node.k \neq k$ then                        ▷ node not found
69:       $result \leftarrow$ false
70:    else
71:       if $val \neq \perp$ then                    ▷ check for logical delete
72:          if CAS$(node.v, val, \perp)$ then        ▷ mark as deleted
73:             remove$(node.prev, node)$             ▷ physical removal
74:             $result \leftarrow$ true
75:       else $result \leftarrow$ false              ▷ logically deleted
76:    return result

77: finish$(insert, k, v, node, val, next, next\_val)_p$:
78:    $result \leftarrow \perp$
79:    if $node.k = k$ then
80:       if $val = \perp$ then                       ▷ check for logical delete
81:          if CAS$(node.v, \perp, v)$ then          ▷ logical insertion
82:             $result \leftarrow$ true
83:       else $result \leftarrow$ false              ▷ not logically deleted
84:    else
85:       $new \leftarrow$ setup_node$(node, next, k, v)$
86:       if CAS$(node.next, next, new)$ then         ▷ insertion
87:          $next.prev \leftarrow new$               ▷ safe
88:          $result \leftarrow$ true
89:    return result
```

is not needed here as *prev* is only used by traversals for backtracking to the list). If either of the CAS operations fails then a concurrent operation has modified the list and $\perp$ is returned (line 89) allowing the traversal to continue.

B. Structural adaptation

The structural adaptation is executed continuously by a dedicated thread, called the *adapting thread*, that repeatedly cleans up the structure by physically removing deleted nodes and adapts the structure by raising or lowering towers appropriately. Even though a failure of the adapting thread does not impact the correctness of our algorithm, a distributed adaptation is discussed in Section VI-B.

1) Physical removal: The first task of the adapting thread lies in physically removing marked deleted nodes of height 1 who were not removed during the delete operation. This task is executed continuously by the adapting thread while traversing the list. The remove operation is more difficult than the abstract logical delete operation as it requires three CAS operations. To illustrate why three CAS operations are necessary, assume that node n with predecessor node *prev* and successor node *next*, is to be removed. If a CAS is performed on *prev.next* in order to remove n by changing the pointer's value from n to *next* then a concurrent insert operation could have added a new node in between n and *next*, leading to a lost update problem [18]. In order to avoid such cases, physical removals are broken into two steps.

In the first step the v field of the node to be removed is CASed from $\perp$ to point to the node itself (line 92). This indicates to other threads that the node is going to be removed. Following this, the removal is completed in a separate help-remove procedure (which might also be called by a concurrent operation performing a traversal).

We encompass lost insert scenarios by using a special marked node, which is inserted into the list directly after the node to be removed using a CAS during the help-remove procedure (lines 102-103). Additionally, during the insert operation, before adding a new node to the list a validation is performed ensuring that neither the predecessor nor the successor node is marked (resulting in the fact that new nodes are never inserted before or after markers), therefore preventing lost inserts (lines 45 and 49 of the do_operation procedure). In order to distinguish a marked node from other nodes it has its *marked* flag set to true and its v field pointing to itself. To complete the removal a CAS is performed on the predecessor's next pointer (line 107) removing both the node and its marker node from the list. This is similar to the process done in Lea's ConcurrentSkipListMap.

2) Raising towers: In the second task of the adapting thread the upper levels of the skip list (i.e. the index-item lists) are modified in order to ensure the $O(\log n)$ expected traversal time. Interestingly, calculating the height of a node cannot be done similarly to traditional skip lists due to the fact that only nodes with a height of 1 are removed. Traditional skip list algorithms favor probabilistic balancing using a random function to calculate the heights of towers as it was more efficient in sequential executions, while here heights are computed deterministically. This deterministic rebalancing is done using the procedure raise-index which is repeatedly called by the

Algorithm 2 Contention-friendly non-blocking skip list – structural adaptation by process p

```
90:  remove(pred, node)_p:
91:    if $node.level = 0$ then                    ▷ only remove short nodes
92:      CAS($node.v, \perp, node$)                ▷ mark for removal
93:      if $node.v = node$ then
94:        help_remove(pred, node)

95:  help-remove(pred, node)_p:
96:    if ($node.val \neq node \vee node.marker$) then
97:      return
98:    $n \leftarrow node.next$
99:    while $\neg n.marker$ do                     ▷ marker to prevet lost inserts
100:     $new \leftarrow$ setup_node($node, n, \perp, \perp$)
101:     $new.v \leftarrow new$
102:     $new.marker \leftarrow$ true
103:     CAS($node.next, n, new$)                  ▷ insert the marker
104:     $n \leftarrow node.next$
105:   if ($pred.next \neq node \vee pred.marker$) then
106:     return
107:   CAS($pred.next, node, n.next$)              ▷ remove the nodes

108: lower-index-level()_p:
109:   $index \leftarrow top$
110:   while $index.down.down \neq \perp$ do
111:     $index \leftarrow index.down$               ▷ get to the $2^{nd}$ lowest level
112:   while $index \neq \perp$ do
113:     $index.down \leftarrow \perp$                ▷ remove the index-level below
114:     $index.node.height \leftarrow index.node.height - 1$
115:     $index \leftarrow index.next$

116: raise-index()_p:
117:   $max \leftarrow -1$
118:   $next \leftarrow top$
119:   while $next \neq \perp$ do                   ▷ add leftmost idx-items to array
120:     $max \leftarrow max + 1$
121:     $first[max] \leftarrow next$
122:     $next \leftarrow next.down$
123:   $inc\text{-}lvl \leftarrow$ raise-nlevel($first[max].node, first[max], 0$)
124:   for ($i \leftarrow max$; $i > 0$; $i \leftarrow i - 1$) do   ▷ traverses indices
125:     $inc\text{-}lvl \leftarrow$ raise-ilevel($first[i], first[i-1], max - i$)
126:   if $inc\text{-}level$ then
127:     $new.down \leftarrow top$                   ▷ allocate an index-item: new
128:     $top \leftarrow new$                        ▷ add a new index-level

129: raise-ilevel(prev, prev-tall, height)_p:   ▷ raise index
130:   $raised \leftarrow false$
131:   $index \leftarrow prev.right$
132:   while true do
133:     $next \leftarrow index.right$              ▷ traverse the list
134:     if $next = \perp$ then
135:       break()
136:     while $index.node.v = index.node$ do
137:       $prev.right \leftarrow next$             ▷ skip removed nodes
138:       if $next = \perp$ then
139:         break()
140:       $index \leftarrow next$
141:       $next \leftarrow next.right$
142:     if ($prev.node.level \leq height$
143:       $\wedge$ $index.node.level \leq height$
144:         $\wedge$ $next.node.level \leq height$) then
145:       $raised \leftarrow true$
146:       $new.down \leftarrow index$             ▷ allocate a index-item: new
147:       $new.node \leftarrow index.node$
148:       $new.right \leftarrow prev\text{-}tall.right$
149:       $prev\text{-}tall.right \leftarrow new$  ▷ raise the tower
150:       $index.node.level \leftarrow height + 1$
151:       $prev\text{-}tall \leftarrow new$
152:     $prev \leftarrow index$                    ▷ continue the traversal
153:     $index \leftarrow index.right$
154:   return raised
```

adapting thread. This procedure starts by setting up an array called *first* so that it contains the first element of every list level. Following this, the raise-nlevel (line 123) procedure is called to raise towers from nodes of height 1 into the index-item list above, followed by raise-ilevel which is called at each index-item list level from the bottom level upwards selecting towers to raise higher one level (lines 124–125).

Each iteration of these procedures traverses the entire list level, during which each time it observes 3 consecutive nodes/towers whose height is equal to that of the level being traversed (lines 142–144) it increments the height of the middle node/tower by 1 (line 150) (Note that raise-nlevel is not included in the pseudo code because it follows the same structure as raise-ilevel with the only difference being that the nodes level is traversed instead of index-item levels). Such a technique approximates the targeted number of nodes present at each level, balancing the structure. Given that physical removals can happen concurrently with the raising of a nodes level, a node that was physically removed might exist as a tower in an index level. When the raise-index-level notices such a situation it simply removes the corresponding index-item (lines 136–141). Following the traversals, if there is at least one node in the highest index level then a new index level is needed (i.e. the last call of raise-index-level returned true), this is done by the adapting thread by simply adding a new index node to the top of the dummy node's tower and modifying the *top* pointer (lines 126–128).

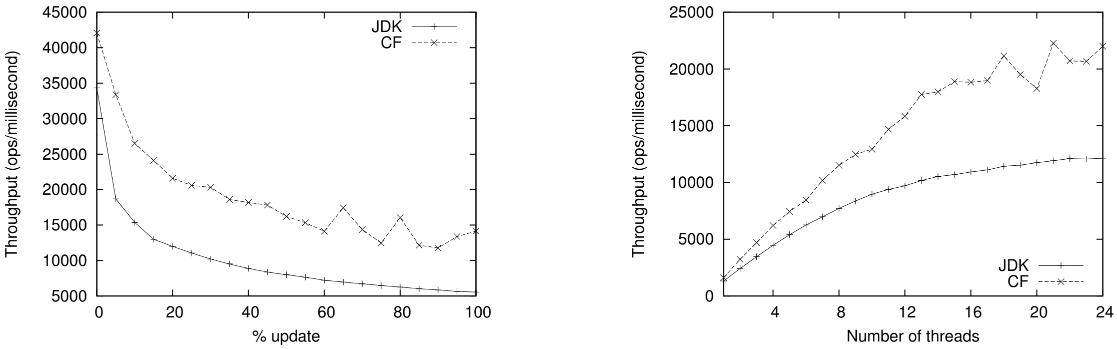
3) Lowering towers: The final task of the adapting thread is due to the fact that only nodes of height 1 (i.e. short towers) are physically removed and is necessary in the case that "too many" tall towers are marked as deleted while most of the short towers between them have been physically removed. If the number of logically deleted towers of height greater than 1 passes some threshold (based on the number of logically deleted nodes and the total number of nodes) then the lower-index-level procedure is called. This procedure simply removes the entire bottom *index-item* list level of the skip list by changing the *down* pointers of each index-item of the level above to ⊥ (line 114). Doing this avoids modification to the taller (i.e. contended) towers in the list and helps ensure there are not too many marked deleted nodes left in the list. Importantly because of this there is no frequent re-balancing of the towers going on, tall towers will remain tall, resulting in less contention at the frequently traversed locations of the structure.

V. Evaluation

Here we compare our skip list to the java.util.concurrent skip list on a multi-core machine. We evaluate their tolerance to contention and their scalability but also dissect the performance of two variants of the contention-friendly skip list and test them under shrink and grow workloads to better assess the reason of the results. We complete our evaluations by comparing the two non-blocking skip lists within the SPECjbb main-memory database benchmark.

A. Settings

The machine used for the tests runs two AMD 12-core processors, comprising 24 hardware threads in total. The tests were done using a microbench with each program thread repeatedly calling insert, remove, and contains operations. The ratio of update operations (insert, remove) to contains is set depending on the test. Insertions and deletions are executed with the same probability so that the data structure size remains constant in expectation with small variations due to concurrency. For each run we averaged the number of executed operations per millisecond over 5 runs of 5 seconds. The five runs execute successively as part the same JVM for



(a) Tolerance to contention of the skip lists (24 threads)

(b) Scalability of the skip lists (20% update)

Fig. 3. Comparison of our CF skip list against the JDK concurrent skip list

the sake of warmup. We used Java SE 1.6.0 12-ea in server mode and HotSpot JVM 11.2-b01.

The JDK skip list is Doug Lea's Java implementation relying on Harris, Michael and Fraser algorithms [9], [13], [8]. It comes with JDK 1.6 as part of the java.util.concurrent package. We compare this implementation to our contention-friendly (CF) skip list as given in Section IV – both implementations are non-blocking.

B. Tolerance to contention

Figure 3(a) depicts the tolerance to contention of the algorithms, by increasing the percent of update operations from 0 to 100 (i.e., between 0% and 50% effective structure updates) with a thread count of 24. The approximate number of elements in the set abstraction is 5 thousand with operations choosing from a range of 10 thousand keys. We can see that the contention-friendly (CF) skip list has significantly higher performance (up to 2.5×) than the Java skip list (JDK) especially at high update ratio, indicating that the former better tolerates contention than the latter.

An interesting result is the performance gain when there is no contention (0% update). As the CF skip list does not have contention hot spots, it can afford maintaining indices for half of the nodes, so that half of the node have multiple levels. In contrast, the JDK skip list maintains the structure so that only one quarter of the nodes have indices, to balance artificially the traversal complexity with the cost induced by hot spots. Not only does our strategy better tolerates contention, but it also improves performance in the absence of contention.

Figure 3(b) compares the performance of the skip list algorithms, run with 20% of update operations (i.e., 10% effective structure updates). Although the JDK skip list scales well with the number of threads, the contention-friendly skip list scales better. In fact, the decoupling of the later allows to tolerate the contention raise induced by the growing amount of threads, leading to a performance speedup of up to 1.8×.

C. On the effect of maintenance and removal

In this section we present additional results in order to better assess the performance benefits of the contention-friendly skip list. For these experiments we include the following two skip list algorithms that are based on the contention-friendly methodology, using certain parts, but not all of it.

- **Non-removal contention-friendly version (CF-NR):** This version of the algorithm does not perform any physical removals. A node that is deleted is only marked as deleted, staying in the skip list forever. This algorithm helps us examine the cost of contention caused by physical removals. A dedicated adapting thread takes care of raising the towers.
- **Non-adapting contention-friendly version (CF-NA):** This version of the algorithm has no dedicated adapting thread. It only uses the selective removal concept of contention friendliness; only nodes of height 1 are physically removed. This version helps us examine the benefits of using an adapting thread. Given that there is no adapting thread, modifications to the upper list levels are done as part of the abstract operations using CAS operations for synchronization. A node's height, like in a traditional skip list, is chosen during the insert operation by a pseudo-random function with the one exception that a height greater than 1 will only be chosen if both of the node's neighbors have a height of 1. This avoids having "too many" tall nodes due to the fact that only nodes of height 1 are physically removed.

Table I depicts the slowdown due to contention by comparing the performance to the non-contended case (0% update ratio) always with 24 threads running. The update ratio is shown at values between 5% and 100% (50% effective). Two set sizes are used, one containing approximately 64 elements and the other containing 64 thousand elements. The range of keys the abstract operations can choose from is either 128 (S) or 128 thousand (L) for the 64 element set size and 128 thousand (S) or 128 million (L) for the 64 thousand element set size. The smaller range allows for higher contention on specific keys in the set while the larger range allows for more variation in the keys in the set.

Considering the 64 element set size, for both ranges the slowdown is much greater using the JDK skip list compared to any of the other algorithms. For the JDK skip list we see up to nearly an 18× slowdown at 100% update with the large range while any of the others is always less than 10×. Between the

TABLE I
SLOWDOWN OF INCREASING UPDATE RATIO % USING OUR CF SKIP LISTS AGAINST THE JDK SKIP LIST WITH A SET SIZE OF 64 OR 64K ELEMENTS AND A SMALL (S) OR LARGE (L) RANGE OF KEYS (24 THREADS).

Update	Range	Set Size	Skip-list Slowdown			
			JDK	CF	CF-NR	CF-NA
5	S	64	2.3	2.1	1.4	2.2
		64k	2.1	1.5	1.3	1.5
	L	64	3	1.6	1.3	2.1
		64k	1.9	1.5	2	1.7
10	S	64	3.4	2.6	1.6	3.1
		64k	2.3	1.5	1.4	1.6
	L	64	4.2	2	1.6	3.1
		64k	2.2	1.7	2.3	1.7
20	S	64	5.1	3.1	2.2	4.3
		64k	2.6	1.7	1.4	1.8
	L	64	6.3	2.6	1.9	3.7
		64k	2.6	1.7	2.3	1.8
50	S	64	9	4.6	2.9	6.2
		64k	3.4	1.8	1.6	1.8
	L	64	11	3.6	3	6.2
		64k	3.5	1.9	3.8	2.3
100	S	64	14	6.4	4.5	9.4
		64k	4.5	1.9	1.6	2.2
	L	64	18	5.2	4	8.7
		64k	5.1	2.2	4.3	2.3
Min			1.9	1.5	1.3	1.5
Max			18	6.4	4.5	9.4
Average			5.3	2.6	2.3	3.4

TABLE II
THROUGHPUT OF INCREASING THREAD NUMBER USING OUR CF SKIP LISTS AGAINST THE JDK SKIP LIST WITH A SET SIZE OF 64K ELEMENTS, AN UPDATE PERCENTAGE OF 10% OR 10% ELEMENTS, AND A SMALL (S) OR LARGE (L) RANGE OF KEYS.

Thds	Range	Update	Skip-list Throughput			
			JDK	CF	CF-NR	CF-NA
1	S	10%	722	738	722	651
		100%	564	569	625	514
	L	10%	779	839	662	654
		100%	561	666	462	515
2	S	10%	1316	1474	1405	1231
		100%	963	1176	1220	902
	L	10%	1367	1532	1126	1245
		100%	1006	1228	851	831
4	S	10%	2321	2878	2749	2364
		100%	1774	2010	2398	1658
	L	10%	2599	3004	1943	2306
		100%	1798	2240	1428	1611
8	S	10%	4389	5249	5179	4161
		100%	3209	4084	4673	3223
	L	10%	4872	5218	3880	4161
		100%	3176	4626	2413	3138
12	S	10%	6280	7941	8028	5853
		100%	4389	6029	7226	4707
	L	10%	6715	7268	5434	6099
		100%	4056	6448	3147	4405
24	S	10%	10279	14332	14303	11047
		100%	5282	11320	12287	8246
	L	10%	10108	13774	9043	10386
		100%	4402	11360	3594	8637
Min			561	569	462	514
Max			10279	14332	14303	11047
Average			3455	4833	3950	3689

CF versions we see that CF-NR provides the best performance which can be explained as follows. As the set size is small, performing marked removals induces much less contention than performing physical removals. CF performs better than CF-NA and JDK, with a maximum slowdown of about 6×.

For the 64 thousand element set size CF shows both good performance and small slowdown while CF-NA shows performance in-between CF and the JDK skip list. CF-NR shows the best performance with the range of 128 thousand elements, but performs poorly when the range is set to 128 million elements. In this case it has a slowdown of over 4× compared to the other CF algorithms which have slowdowns of around 2×. This must be due to the very large range: since CF-NR does not perform any physical removal the number of logically deleted nodes in the skip list grows so large that the cost of traversing them become significant. In particular, we saw in this benchmark a skip list of up to 6 million nodes.

D. Evaluating scalability

Table II displays the scalability of the algorithms by showing the effect of increasing the number of threads from 1 to 24. The tests were done with 10% and 100% update ratios with a set size of approximately 64 elements or 64 thousand elements. The small set size was tested with a range of 128 (S) and 128 thousand (L) keys while the large set size used a range of 128 thousand (S) and 128 million (L).

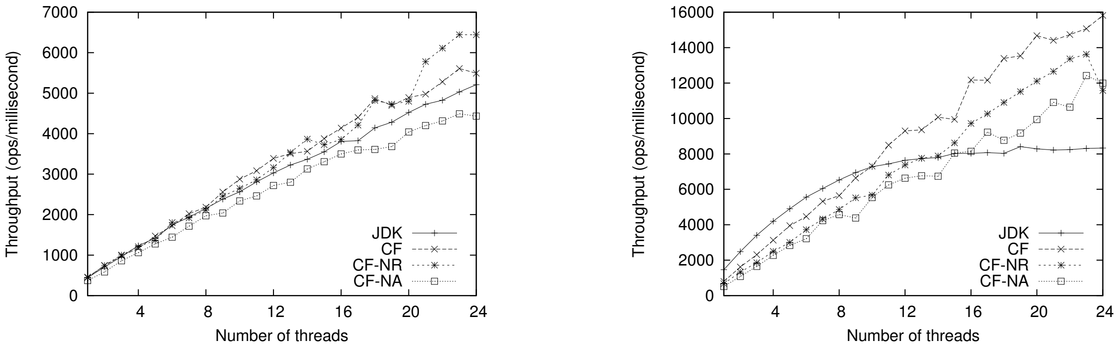
First consider the 64 element set size. At 10% update all algorithms show good scalability for both ranges with CF showing the best performance of all algorithms. Things get more interesting at 100% update. For both range options (S and L) the JDK skip list starts loosing scalability after 12 cores, while CF performs well all the way up to 24-cores. For example, when using the large range (L) the JDK skip list has a speedup of 1.08× when going from 12 to 24 cores, while CF has a speed up of 1.76×. CF-NA scales well, but does not perform quite as well as CF, showing the advantage of having the adapting thread. CF-NR performs well when the range of 128 thousand is used, but performs poorly when the range is set to 128 million, again this is due to not physically removing nodes resulting in a very large list size.

E. Quality of the structural adaptation

The purpose of Figures 4(a)-4(b) is to test whether or not the adapting thread is effective when the number of elements in the set changes drastically. This is done by the following: In the grow benchmark the size of the set starts at 0 elements and grows until a size of 500 thousand elements, while the shrink benchmark starts with a set of size 500 thousand elements and ends with 2,500 elements. Like before the duration of these benchmarks is 5 seconds and they use 24 threads. Both benchmarks are executed with a 50% update ratio.

In the grow benchmark (Figure 4(a)) all algorithms show good scalability with a small performance advantage going to the algorithms with adapting threads (CF, CF-NR). This is likely due to the fact that these algorithms do not require synchronization operations on the towers.

In the shrink benchmark (Figure 4(b)) we see that the JDK skip list performs best at small thread counts while the contention-friendly algorithms show better scalability. Due to the decreasing list size the CF skip list calls the



(a) Grow benchmark

(b) Shrink benchmark

Fig. 4. Comparison of the scalability of our CF skip lists against the JDK concurrent skip list using the grow and shrink benchmarks (20% updates)

lower-index-level procedure on average 4 times per run of the benchmark and at the end of the benchmark the skip list contains around 15 thousand marked deleted nodes. Here lower-index-level is called by the adapting thread when it discovers that there are at least 10 times more marked deleted nodes than non-marked deleted ones. This number can be tuned so that the procedure is called more or less often, resulting in a skip list with a larger or smaller amount of marked deleted nodes.

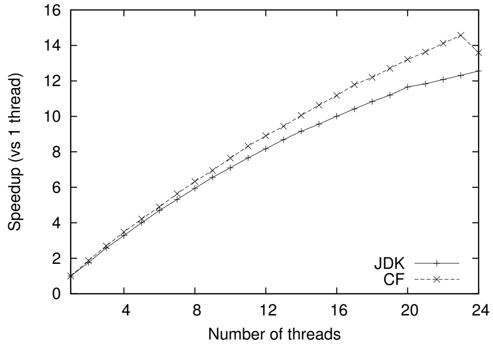


Fig. 5. Speedup of the CF skip list and the JDK concurrent skip list using a modified version of the SPECjbb 2005 benchmark

F. SPECjbb 2005 as a main-memory database benchmark

SPECjbb 2005 [16] is a highly scalable Java server benchmark designed to test different components, mainly focusing on the JVM. It is a multi-threaded benchmark but threads share very little data. The benchmark is based on the emulation of a three-tier client/server system using an in-memory database backed by Java collections. In addition to accessing a concurrent collection the benchmark also performs (among other things) Java BigDecimal computations, XML processing, and accesses to thread local collections. The collections implement the Java Map interface so they can be swapped for the JDK skip list or the CF skip list. Unfortunately in the default benchmark these collections are all thread local but not accessed concurrently. For the sake of concurrency, we replaced certain thread local collections (specifically the collections used to store orders and order history) with a single concurrent collection (a similar technique was used in [1] to test the scalability of transactional memory). It should be noted that this took a fair bit of modification so the results here should not be compared to other SPECjbb results, our goal here is to test if contention-friendly data structures show an improvement when they a small part of a larger program.

The scalability results of running the modified benchmark from 1 to 24 applications threads are shown in Figure 5. The first thing that should be noticed is that both skip list implementations show very good scalability. At 23 application threads the JDK skip list shows a 12.3× speed-up over a single core, while the CF skip list shows a 14.6× speed-up. There is obviously very little contention on the concurrent map, likely comparable to 1% or 2% update ratio of the microbenchmark as at 5% we are already seeing large slowdowns (Table I at 24 threads). This is probably due to the fact that the accesses to the concurrent map are only part of the program that is performing many other actions. Even so, we do see a separation between the speedups of the skip lists, with increasingly higher speedups for the CF skip list as the core count approaches 23. At 23 cores the JDK skip list version performs 747730 business operations per second (bops), the instrument of measure for SPECjbb performance, while the CF version performs 777662 bops. Overall this is a small performance difference, yet it is unsurprising given the already good scalability of the benchmark and the fact that the concurrent map accesses are only one piece of the benchmark.

Interestingly at 1 core the JDK skip list version has a small, but significant (7000 bops) performance advantage over the CF one which diminishes until 19 applications threads when the CF skip list takes over in performance. We believe this to be due to the fact that the JDK skip list is optimized for the JVM, minimizing the cost of loading the skip-list back to a thread's memory after it had been executing other tasks. Unsurprisingly the CF version takes a large performance hit when going from 23 to 24 applications threads because at that point the adapting thread and program threads must compete for processor time. Such observations tend to indicate that the CF skip list is really beneficial when there are less application threads running than available cores.

VI. Extending the CF Skip List

A. Memory management

Nodes that are physically removed from the data structures must be garbage collected. Once a node is physically removed it will no longer be reachable by future operations.

Concurrent traversal operations could be preempted on a removed node so the node cannot be freed immediately. In languages with automatic garbage collection these nodes will be freed when all preempted traversals continue past this node. If automatic garbage collection is not available then some additional mechanisms can be used. One simple possibility is to provide each thread with a local operation counter and a boolean indicating if the thread is currently performing an abstract operation or not. Then any physically removed node can be safely freed as long as each thread is either not performing an abstract operation or if it has increased its counter since the node was removed. This can be done by the adapting thread. Other garbage collection techniques can be used such as reference counting described in [4].

B. Distributing the structural adaptation

The algorithm we have presented exploits the multiple computational resources available on today's multicore machines by having a separate adapting thread. It could be adapted to support multiple adapting threads or to make each application thread participate into a distributed structural adaptation (if for example computational resources get limited). Although this second approach is appealing to maintain the asymptotic complexity despite failures, it makes the protocol more complex.

To keep the benefit from the contention-friendliness of the protocol, it is important to maintain the decoupling between the abstract modifications and the structural adaptations. Distributing the tasks of the adapting thread to each application threads should not force them to execute a structural adaptation systematically after each abstract modification. One possible solution is that each application thread tosses a coin after each of its abstract modification to decide whether to run a structural adaptation. This raises an interesting question on the optimal proportion of abstract modifications per adaptation.

Another challenge of having concurrent structural adaptation is to guarantee that concurrent structural adaptations execute safely. This boils down into synchronizing the higher levels of the skip list by using CAS each time a pointer of the high level lists is adapted. An important note is that given the probability distribution of nodes per level in the skip list, the sum of the items in the upper list levels is approximately equal to the number of nodes in the bottom list level. On average the amount of conflicts induced by the skip list with a distributed adaptation could be potentially twice the one of the centralized adaptation. This exact factor depends, however, on the frequency of the distributed structural adaptation.

Finally, to distribute the structural adaptation each thread could no longer rely on the global information regarding the heights of other nodes. To recover to the probability distribution of item to levels without heavy inter-threads synchronization, a solution would be to give up the deterministic level computation adopted in the centralized version and to switch back to the traditional probabilistic technique: each application thread inserting a new node would simply choose a level ℓ with probability $2^{-O(\ell)}$.

In the case where there are additional resources available it might be interesting to assign multiple threads to separate adapting tasks. For example one thread could be responsible for choosing the heights of the nodes, with another responsible for the upper level modifications, and a third responsible for physical deletions.

VII. Conclusion

Multicore programming brings new challenges, like contention, that programmers have to anticipate when developing every day applications. We explore the design of a contention-friendly and non-blocking skip list, keeping in mind that contention is an important cause of performance drops.

As future work, it would be interesting to extend the interface to support additional methods.

References

[1] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *PPoPP*, pages 56–67, 2007.

[2] T. Crain, V. Gramoli, and M. Raynal. A contention-friendly, non-blocking skip list. Technical Report RR-7969, IRISA, 2012.

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

[4] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. In *PODC*, pages 190–199, 2001.

[5] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[6] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, volume 5805 of *LNCS*, pages 93–107, 2009.

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

[8] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University, September 2003.

[9] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 2180 of *LNCS*, pages 300–314, 2001.

[10] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *SIROCCO*, volume 4474 of *LNCS*, pages 124–138, 2007.

[11] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[12] D. Lea. JSR-166 specification request group. http://g.oswego.edu/dl/concurrency-interest.

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

[14] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *PODS*, pages 170–176, 1987.

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

[16] Standard performance evaluation corporation, SPECjbb2005 benchmark, 2005. http://www.spec.org/jbb2005/.

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

[18] J. D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, May 1995.