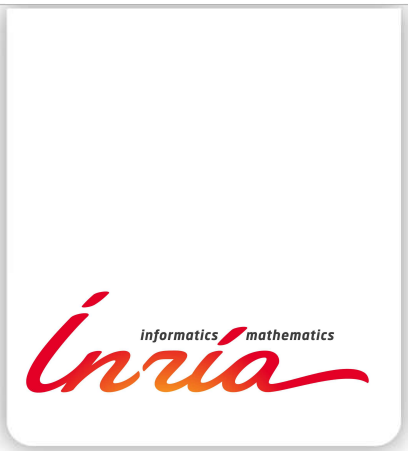
A Contention-Friendly, Non-Blocking Skip List

Tyler CRAIN, Vincent GRAMOLI, Michel RAYNAL
tyler.crain@irisa.fr, vincent.gramoli@sydney.edu.au, raynal@irisa.fr

RESEARCH REPORT
N° 7969
May 2012
Project-Team ASAP

ISSN 0249-6399 ISRN INRIA/RR--7969--FR+ENG



A Contention-Friendly, Non-Blocking Skip List

Tyler CRAIN*, Vincent GRAMOLI†, Michel RAYNAL‡*
tyler.crain@irisa.fr, vincent.gramoli@sydney.edu.au,
raynal@irisa.fr

Project-Team ASAP

Research Report n° 7969 — May 2012 — 18 pages

Abstract: This paper presents a new non-blocking skip list algorithm. This algorithm limits the high contention induced by today's multicore environments to come up with a more efficient alternative than existing ones.
Data structures like skip lists are generally constrained to guarantee a big-oh step complexity even in the presence of concurrency. By contrast our idea is to guarantee the big-oh complexity only in the absence of contention and limits the contention when concurrency appears. The key concept lies in dividing update operations within an *eager abstract modification* that returns rapidly for efficiency reason and a *lazy selective adaptation* that may be postponed to diminish contention.
The skip list algorithm is proved correct by proving the linearizability of a map (i.e., dictionary) abstraction it implements. It is evaluated experimentally in Java and compared to the ConcurrentSkipListMap on a multi-core machine. In particular, the skip list is up to 2.5× faster than the Java concurrent skip list.

Key-words: Lock-based, Lock-free, Eager abstract modification, Lazy structural adaptation

* IRISA, Université de Rennes 35042 Rennes Cedex, France
† University of Sydney
‡ Institut Universitaire de France

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Un concurrent non-bloquant listes á saut (skip list)

Résumé : Ce rapport présente une approche méthodologique pour les structures de recherche concurrentes avec des applcations aux non-bloquant listes á saut (skip list).

Mots-clés : mémoire transactionnelle, structures de données concurrente

1 Introduction

Multicore architectures are changing the way we write programs. Not only are all computational devices turning multicore thus becoming inherently concurrent, but tomorrow's multicore will embed a larger amount of simplified cores to better handle energy while proposing higher performance, a technology also known as *manycore* [1]. Programmers must thus change their habits to design new concurrent data structures that would be otherwise the bottlenecks of modern every day applications.

The big-oh complexity, which indicates the worst-case amount of converging steps necessary to complete an access, used to prevail in the choice of a particular data structure algorithm running in a sequential context or with limited concurrency. Yet contention has now become an even more important factor of performance drops in today's multicore systems. For example, some concurrent data structures are even so contended that they cannot perform better than bare sequential code, and exploiting additional cores simply make the problem worse [14].

A skip list is a probabilistic data structure to store and retrieve in-memory data in an efficient way, especially used in database systems. In short, a skip list is a linked structure that diminishes the linear big-oh complexity of a linked list with elements having additional shortcuts pointing towards other elements located further in the list [13]. These shortcuts allows operations to complete in $O(\log n)$ steps in expectation. The drawback of employing shortcuts is however to require additional maintenance each time some data is stored or discarded. This causes contention overheads on concurrent skip lists by increasing the probability of multiple *threads* (or processes) interfering on the same shared data. This typically translates into significant performance losses on machine with a large amount of cores.

In the light of the impact of contention on performance, we propose a *Contention-Friendly (CF)* non-blocking skip list that accommodates contention in modern multi-/many-core machines without relaxing the correctness of the abstractions. To this end, we argue for a genuine decoupling of each updating access into an eager abstract modification and a lazy structural adaptation that is selective.

- The *eager abstract modification* consists in modifying the abstraction while minimizing the impact on the skip list itself and returning as soon as possible for the sake of responsiveness.

- The *lazy selective adaptation*, which can be deferred until later, aims at adapting the skip list structure to these changes by re-arranging elements or garbage collecting deleted ones.

More specifically, the aforementioned decoupling translates into splitting an element insertion into the insertion phase at the bottom level of the skip list and the structural adaptation responsible for updating pointers at its higher levels, and an element removal into a logical deletion marking phase and its physical removal and garbage collection.

Additionally, our contention-friendly skip list is *non-blocking*, ensuring that the system as a whole always makes progress.[1] Shortening operations so that they return just after the abstract access diminishes their latency whereas postponing the structural adaptation to avoid temporary load bursts and making it selective to avoid the localized hot-spots helps diminish the contention but also potential starvation.

We prove our algorithm correct and we compare its performance against the Java adaptation by Lea of Harris, Michael and Fraser's algorithms [7, 8, 11]. This implementation is probably one of the mostly used non-blocking skip lists today and is distributed within the Java Development Kit. Our results observed on our 24-core AMD machine shows a 2.5× speedup.

Section 2 describes the related work. Section 3 depicts how to make a skip list contention-friendly. Section 4 describes in details our contention-friendly non-blocking skip list algorithm. Section 5 presents the experimental results and Section 6 concludes. Appendix A depicts additional experimentations and Appendix B shows our algorithm correct.

2 Related Work

Decoupling each data structure modification into multiple tasks has proved beneficial for memory management [4] and efficiency [2, 12], yet this idea was essentially applied to balanced trees but not to diminish contention in skip lists.

[1] Note that we prefer the term *non-blocking* to the term *lock-free* to denote our targeted progress guarantee. We found it helpful in distinguishing our approach from blocking techniques that do not use locks explicitly.

Tim Harris proposed to mark elements for deletion using a single bit prior to physically removing them [8]. This bit corresponds typically to the low-order bit of the element reference that would be unused on mostly modern architectures. The decoupling into a logical deletion and a physical removal allowed Harris to propose a non-blocking linked list using exclusively 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 [11] and by Keir Fraser to develop a non-blocking skip list [7].

Doug Lea adapted these algorithms to propose a non-blocking skip list implementation in Java [10]. 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. The structure comprises one tower per element whose level is determined by a pseudo-random function such that the probability for a tower to have level ℓ is $2^{-O(\ell)}$. A tower of level ℓ comprises $\ell - 1$ *index-items*, one above the other, under which a *node* is used 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 [17] of constructing non-blocking dictionaries using linked structures. They propose to complement Valois' thesis by specifying a practical non-blocking skip list that implements a dictionary abstraction [15]. 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 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 proposed a non-blocking skip list algorithm whose towers are linked through a doubly linked list [6]. In addition to the original skip list structure [13], it requires backward links to let a traversal potentially backtrack. They also use the logical deletion mechanism and a tower is deleted by first having its bottommost node marked for deletion, 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.

Our non-blocking skip list algorithm uses the same logical deletion technique and the removal process requires some help from another traversal as it is the case in previous skip lists. The main novelty is the decoupling of the abstract modification from the selective structural adaptation to achieve contention-friendliness. In particular, the physical removal applies selectively to towers of low levels to avoid a contention increase at hot spots and some insertions can be accelerated by being done logically. Although it could be distributed, our current structural adaptation is executed by a single thread. This allows us to design a non-blocking skip list in a simpler way than previous approaches. In particular, the adaptations require synchronization only when accessing nodes, at the bottom level.

Finally, transactional memories can be used to implement non-blocking skip list, however, they may restrict skip list concurrency [7] or block [5]. Note that our notion of contention-friendliness differs from Gadi Taubenfield's contention-sensitivity that was applied to queues [16] as the latter aims at executing an efficient path before switching to a lock-based one when contention raises.

3 Towards Contention-Friendliness

In this section, we give an overview of the technique to make the skip list contention-friendly. The crux lies in modifying the traditional skip list structure without relaxing the abstraction or their correctness. Our contention-friendly skip list aims at implementing a correct *map*, or dictionary, abstraction as it represents a common example for storing key-value pairs. The correctness criterion ensured here is linearizability [9].

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) Inserting horizontally in the skip list

(b) Adapting vertically the skip list structure

Figure 1: Decoupling the eager abstraction insertion from the lazy selective adaptation

3.1 Eager abstract modification

Previous skip lists maintain the node per level distribution so that the probability of a node i to have level ℓ is $\Pr[level_i = \ell] = 2^{-O(\ell)}$, hence each time the abstraction is updated, the invariant is checked and the structure is accordingly adapted as part of the single operation. While 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 contention-friendly skip list only the bottom level is modified and the additional structural modification is postponed until later. In Appendix B we show that this abstract modification is sufficient to guarantee linearizability. This decoupling avoids an insertion to update up to $O(\log n)$ levels, this reduces contention and makes the update cost of each operation more predictable.

As an example, assume we aim at inserting an element with key 12 in a skip list. Our insertion consists in updating only the bottom most level of the structure by adding a new node to this level, 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 linking this same node at higher levels is deferred until later.

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

3.2 Lazy selective structural adaptation

It is important to guarantee the logarithmic complexity of accesses when there is no contention in the system. Hence when contention stops, the structure needs to be adapted by setting the next pointers at upper levels of the skip list. Figure 1(b) depicts the structural adaptation corresponding to the insertion of node 12: the insertion at a higher level of the skip list is executed as a structural adaptation (separated from the insertion), which produces eventually a good distribution of nodes among levels.

Laziness to avoid contention bursts. The structural adaptation is *lazy* because it is decoupled from the abstract modifications and executed by independent threads. 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 whose prominent one is to enable merging of multiple adaptations in one simplified step: only one traversal is sufficient to adapt the structure after a bursts of abstract modifications. Another interesting aspect is that it gives a chance to insertion to execute faster: if the element to be inserted is logically deleted, then the insertion simply needs to logically insert by unmarking it as logically deleted. This avoids the insertion to allocate a new node and to write its value in memory.

Selectivity to avoid contention hot-spots. The abstract modification of a removal simply consists of marking the nodes as deleted without modifying the actual structure. The subsequent structural adaptation selects for removal the nodes whose removal would induce the least contention.

A removal of a node with a high level, say the one with value 36 in Figure 1(b), would typically induce more contention than the removal of a node with a lower level, say the one with value 62 spanning a single level. The reason is twofold, first removing a node spanning ℓ levels boils down to updating $O(\ell)$ pointers, hence removing node with value 36 requires to update 3 pointers while node with value 62 requires to update $O(1)$ pointers, second, the organization of the skip list implies that the higher level pointers are traversed more frequently, hence

the removal of 36 typically conflicts with every operation concurrently traversing this structure whereas the next pointer of 62 is unlikely to be accessed by a large number concurrent traversals. In the next section, we present an algorithm that removes only towers of height 1.

4 The Non-Blocking Skip List

In this section, we present our contention-friendly non-blocking skip list. Section 4.1 describes the abstract modifications as well as the contains operation. Section 4.2 describes the structural adaptation that is repeatedly executed by a single thread for the sake of simplicity. Section 4.3 gives the intuition of the non-blocking guarantee of our algorithm. Section 4.4 describes the garbage collection. Section 4.5 discusses the distribution on the adaptation on multiple threads. The correctness proof is deferred to the Appendix B.

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 ∈ ℕ, the node key
4:      v, the node's value, a value of ⊥ indicates
5:        the node is logically deleted
6:      marker ∈ {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: delete(k)_p:
13:   node ← traverse-tower(top, k)
14:   while true do
15:     node ← get-next-node(node, k)
16:     if node.k ≠ k then
17:       result ← false
18:       break()
19:     else
20:       v ← node.v
21:       if (v ≠ ⊥ ∧ v ≠ node) then
22:         if CAS(node.v, v, ⊥) then // compare-and-swap
23:           result ← true
24:           break()
25:       else
26:         result ← false
27:         break()
28:   return result

29: insert(k, v)_p:
30:   node ← traverse-tower(top, k)
31:   while true do
32:     node ← get-next-node(node, k)
33:     if node.k = k then
34:       if node.v = ⊥ then // logical insertion
35:         if CAS(node.v, ⊥, v) then // compare-and-swap
36:           result ← true
37:           break()
38:       else
39:         result ← false
40:         break()
41:     else
42:       new ← setup_node(node, k, v)
43:       next ← new.next
44:       if next.val ≠ next then // logical insertion
45:         if CAS(node.next, next, new) then // compare-and-swap
46:           next.prev ← new
47:           result ← true
48:           break()
49:   return result

50: State of index-item:
51:   item a record with fields:
52:     right, pointer to the next
53:       item in the SkipList
54:     down, pointer to the IndexItem
55:       one level below in the SkipList
56:     node, pointer a node in the list
57:       at the bottom of the SkipList

58: setup-node(node, k, v)_p:
59:   new.k ← k; new.v ← v // allocate a node called new
60:   new.prev ← node
61:   new.next ← node.next
62:   return new

63: contains(k)_p:
64:   item ← traverse-tower(top, k) // find the right tower
65:   node ← get-next-node(item, k) // find the right node
66:   result ← false
67:   if node.k = k then
68:     v ← node.v
69:     if (v ≠ ⊥ ∧ v ≠ node) then
70:       result ← true
71:   return result

72: traverse-tower(item, k)_p:
73:   while true do
74:     nitem ← item.right
75:     if nitem.node.k > k then
76:       nitem ← item.down
77:       if nitem = ⊥ then
78:         result ← item.node
79:         break()
80:     else if nitem.node.k = k then
81:       result ← item.node
82:       break()
83:     item ← nitem
84:   return result

85: get-next-node(node, k)_s:
86:   while true do
87:     while node.v = node do node ← node.prev  // backtack
88:     next ← node.next
89:     if (next ≠ ⊥ ∧ next.v = next) then
90:       help-remove(node, next) // help the removal
91:       next ← node.next
92:     if (next = ⊥ ∨ next.k > k) then
93:       result ← node
94:       break()
95:     node ← next
96:   return result
```

Figure 1 depicts the algorithm of the eager abstract operations while Figure 2 depicts the algorithm of the lazy selective adaptation. The skip list algorithm is *non-blocking* meaning that there is always at least one thread makes progress after a sufficiently long amount of time. In particular, only CAS operations are used for synchronization. The bottom level of the skip list is made up of a doubly linked list of nodes as opposed to the Java ConcurrentSkipListMap. Each node has a *prev* and *next* pointer, a key k, a value v, an integer *level* indicating the

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

```
97:  remove(pred, node)_p:
98:    result ← false
99:    if node.level = 0 then
100:     CAS(node.v, ⊥, node) // compare-and-swap
101:     if node.v = node then
102:       help_remove(pred, node)
103:       result ← true
104:   return result

105: help-remove(pred, node)_p:
106:   if (node.val ≠ node ∨ node.marker) then
107:     return
108:   n ← node.next
109:   while ¬n.marker do
110:     new ← setup_node(node, ⊥, ⊥)
111:     new.v ← new
112:     new.marker ← true
113:     CAS(node.next, n, new) // compare-and-swap
114:     n ← node.next
115:   if (pred.next ≠ node ∨ pred.marker) then
116:     return
117:   CAS(pred.next, node, n.next) // compare-and-swap

118: lower-index-level()_p:
119:   index ← first[2].next
120:   while index ≠ ⊥ do
121:     index.down ← ⊥
122:     index.node.height ← index.node.height − 1
123:     index ← index.next
124:   // Update the index of the first array

125: raise-index-level(i)_p:
126:   prev-tall ← first[i + 1]
127:   index ← first[i]
128:   while true do
129:     next ← index.right
130:     if next = ⊥ then
131:       break()
132:     prev ← index.prev
133:     if (prev.node.level ≤ i
134:       ∧ index.node.level ≤ i
135:         ∧ next.node.level ≤ i) then
136:       // Allocate a new index-item called new
137:       new.down ← index
138:       new.node ← index.node
139:       new.right ← prev-tall.right
140:       prev-tall.right ← new
141:       index.node.level ← i + 1
142:       prev-tall ← new
143:     index ← index.right
```

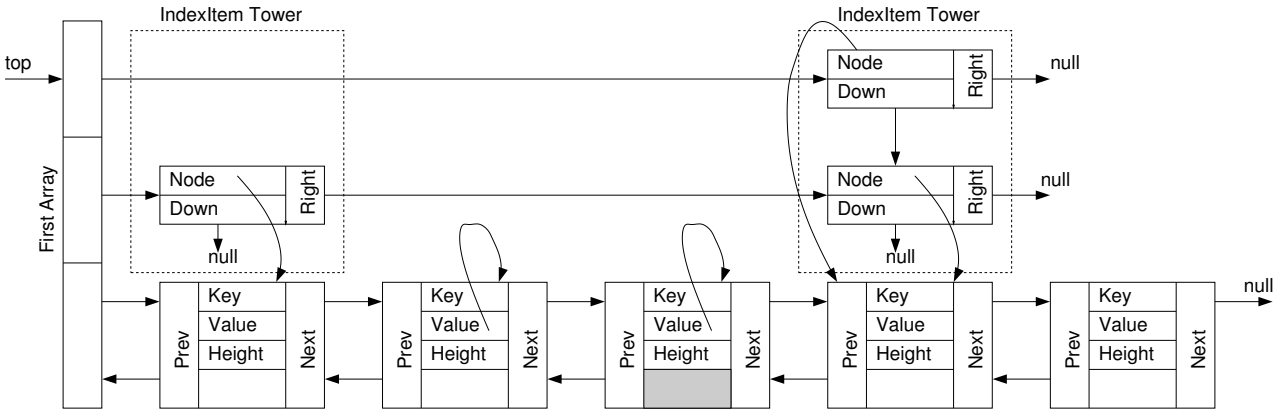


Figure 2: Skip list structure

number of levels of linked lists this node has, a *marker* flag indicating whether or not the node is a marker (used during removals).

We use the logical deletion technique [8] by nullifying the v field used to hold the value associated with the key of the node. If $v = \perp$, then we say that the node is logically deleted. 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 $n.v = n$).

The upper levels are made up of singly linked lists of *index-items*. Each of these items has a *next* pointer, pointing to the next item in the linked list, a *down* pointer, pointing to the linked list of IndexItems one level below (the bottom level of IndexItems have $\perp$ for their *down* pointers), and a *node* pointer that points to the corresponding node at the bottom of the skip list.

A per structure array of pointers called *first* is also kept that points to the first element of each level of the skip list. The pointer *top* points to the first element of the highest index of the list, all traversals start from this pointer. Figure 4 shows the structure of the contention friendly skip list where the third node is in the process of being removed and the fourth node is a marker.

4.1 Abstract operations

The contains, insert, and delete operations start by traversing the towers using the traverse-tower procedure that traverses the towers similar to a sequential skip list algorithm, moving forward in the list until reaching a node with a larger key than k and then moving down a level. If a node with key k is found then that node is returned immediately, otherwise the operations continues until the bottom of the tower is reached, returning the node of the tower it stops at.

The traversal continues on the bottom list level using the get-next-node procedure. The main differences between this traversal and a sequential algorithm is due to concurrent removals. If a node is encountered in the list that has been marked to be removed (line 89) then the help-remove procedure is called (line 90) or if a node is encountered that has already been removed then the *prev* pointers are used to backtrack into the list.

During the contains operation, if a node with key k is found its value is read (lines 67-69) and either true or false is returned depending on the observed value; if no node with key k is found false is returned.

During a delete operation, if a node with key k is found that is neither removed nor marked deleted (checked on line 21) then a CAS is performed to try marking the node as deleted (line 22).

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 deleted (lines 34) by unmarking it (lines 35). If no node with key k is found then the insert operation allocates "physically" a new node (line 42) before adding it to the list by performing a CAS operation on the next pointer of the predecessor (line 45). Note that existing skip list algorithms cannot exploit logical insertions as each logical deletion is traditionally followed by a physical removal.

During both insert and delete operations if a CAS operation fails (due to a concurrent modification) then the get-next-node procedure is called again, starting the traversal from the node where the CAS failed.

4.2 Structural adaptation

The structural adaptation is executed repeatedly by a dedicated thread, called *adapting thread*. Its first task is to physically remove nodes marked as deleted who have a height of 1. This is done after a successful delete operation, as well as in the adapting thread. The remove operation is more difficult than the abstract operations as it requires three CAS operations. The reason is that a node cannot be safely removed from the list using just one CAS. Consider a node n to be removed that has predecessor and successor nodes *prev* and *next*. If a CAS is performed on *prev.next* removing *node* by changing the pointer's value from *node* to *next* then a concurrent insert operation could have added a new node in between *node* and *next*, leading to a lost update problem [17]. In order to avoid such cases, physical removals are broken into two steps.

First, the v field of the node to be removed is CASed from $\perp$ to point to the node itself on line 100 of the remove operation. This indicates to other threads that the node is going to be removed. Then, 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 with a CAS just after the node to be removed (lines 112-113), similar to Lea's ConcurrentSkipListMap. In order to distinguish a marked node from other nodes it has its *marked* flag set to true and its v field points to itself. Additionally, a validation checks that neither the predecessor nor the successor node is marked before inserting a new node. (lines 87 and 92 of the get-next-node procedure and on line 44 of the insert operation). To complete the removal a CAS is performed on the predecessor's next pointer (line 117) removing the node and its marker from the list.

A second task of the adapting thread is to modify the upper levels of nodes in order to ensure the $O(\log n)$ expected traversal time. Since neither removals nor insertions are done as they are in traditional skip lists, calculating the height of a node must also be achieved differently. Existing algorithms call a random function to calculate the heights of nodes, here they are done deterministically while considering that the fundamental structure of a skip list is not designed to be perfectly balanced (as it would be too costly) but rather probabilistically balanced. This is done using the procedure raise-index-level, which is called at each tower level from the bottom level going upwards. Each iteration traverses an entire list level, where each time it observes 3 consecutive nodes whose height equals this level (line 133–135), it raises the level of the middle node by 1 (line 141). Such a technique approximates the targeted number of nodes present at each level, balancing the structure.

The final task of the adapting thread is due to the fact that only towers of height 1 are physically removed and is necessary in the case that "too many" tall nodes are marked as deleted. If the number of nodes of levels greater than 1 that are logically deleted passes some threshold then the lower-index-level procedure is called which

removes the entire bottom *index-item* level of the skip list by changing the *down* pointers of the level above to $\perp$ (line 122). Doing this avoids modification to the taller nodes in the list and helps ensure there are not too many marked deleted nodes left in the list. There are no frequent re-balancing going on to the tower, tall nodes will stay tall nodes meaning less contention at the frequently traversed locations of the structure.

The adapting thread continually traverses the structure repeating these structural adaptation procedures as necessary while also physically removing appropriate nodes that were not successfully removed by a delete operation.

4.3 Non-Blocking

In order for an algorithm to be considered non-blocking at least one thread must make progress after a sufficiently long amount of time. Since none of the operations use locks or any blocking operations the algorithm can be proven to be non-blocking by showing that a thread can only be stuck infinitely in a loop if there is at least one other thread making progress. Many of these loops traverse the skip list following the *next*, *prev* pointers of the nodes or the *down*, *right* pointers of the IndexItems. In Appendix B we show that the skip list is valid which requires that nodes are sorted in ascending order by their keys. Therefore these traversals can only loop infinitely if there is either an infinite number of nodes being added/removed from the list concurrently.

First we will consider the while loop in the get-next-index procedure. Iterations of this loop traverses forward and down levels of IndexItems.

The help-remove procedure has a while loop on lines 109–114, this loop compares and swaps a marker node into the list, looping until a marker is added successfully. In order for the addition of a marker node to fail an infinite number of times there must be an infinite number of concurrent successful insert operations.

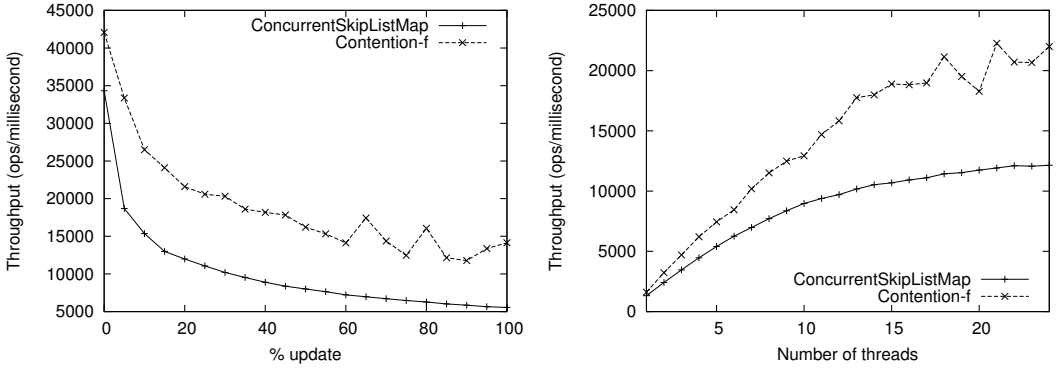
In the get-next-node procedure there is an inner while loop that traverses backward in the list (lines 87–87) and an outer level loop that traverses forward in the list (lines 86–95). An infinite traversal in the inner loop would require an infinite number of node removals which would also require and infinite number of inserts. An infinite number of outer loop iterations would require either a infinitely long list (which would mean an infinite number of inserts) or an infinite number of invocations of the inner while loop. The inner while loop is only invoked when the outer traversal reaches a node in the process of being removed. Therefore in order for the inner loop to be invoked an infinite number of times the outer traversal must encounter one or more marked removed nodes in the list an infinite number of times. Each iteration of the outer loop calls the help-remove procedure, which either successfully physically removes a node from the list or fails due to a concurrent operation succeeding. If it succeeds an infinite number of times then there must be an infinite number of delete operations and if it fails an infinite number of times then there must be an infinite number of concurrent operations modifying the list structure (be it inserts or removals), in either case at least one other thread is making progress.

Both the insert and delete operations contain a while loop. In both operations an iteration of the while loop will always exit unless it calls a CAS that fails. Both operations might perform a CAS on the *v* field of a node (line 35 of insert and line 22 of delete), but this CAS will only fail due to a concurrent insert or delete successfully performing a CAS on this field. The insert operation might perform a CAS on line 45 changing a node's pointer, this CAS can only fail due to a concurrent successful insert or removal. Therefore an infinite loop in these operations is only possible due to infinite number of successful operations by at least one other thread.

4.4 Garbage collection

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 as soon as all preempted traversals continue past this node. If automatic garbage collection is not available then some additional mechanisms can be used. One possibility is to provide each thread with a local operation counter and a boolean indicating if the tread 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 [3].



(a) Tolerance to contention of the non-blocking skip lists

(b) Scalability of the non-blocking skip lists

Figure 3: Comparison of our contention-friendly non-blocking skip list against the JDK concurrent skip list (ConcurrentSkipListMap)

4.5 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 approach is appealing to, for example, to maintain the big-oh 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. Instead, each application thread could toss 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.

The other challenge 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 $2^{-O(\ell)}$.

5 Evaluation

Here we compare our skip list to the java.util.concurrent skip list on a multi-core machine. Additional experiments are deferred to Appendix A. The machine is an AMD with two 12-core processors, comprising 24 hardware threads in total. For each run we averaged the number of executed operations per millisecond over 5 runs of 5 seconds. Thread counts are from 1 to 24 and the five runs execute successively as part the same JVM for the sake of warmup. We used Java SE 1.6.0 12-ea in server mode and HotSpot JVM 11.2-b01.

The approximate number of elements in the set abstraction is 5 thousand with operations choosing from a range of 10 thousand keys, implying that each update operation successfully modify the data structure 50% of the time. As insertions and deletions are executed with the same probability the data structure size remains constant

in expectation.

The ConcurrentSkipListMap is Doug Lea's Java implementation relying on Harris, Michael and Fraser algorithms [7, 8, 11]. It comes with JDK 1.6 as part of the java.util.concurrent package. We compare this implementation to our contention-friendly skip list as given in Section 4—both implementations are non-blocking.

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). We can see that the contention-friendly (Contention-f) skip list better tolerates contention than the ConcurrentSkipListMap which results in significantly higher performance.

An interesting result is the gain of using our skip list at 0% update: since the contention-friendly skip list tolerates high contention, it can afford maintaining indices for half of the nodes, so that half of the node have multiple levels. In contrast, ConcurrentSkipListMap maintains the structure so that only one quarter of the nodes have indices, in an attempt to reduce the contention when updates come into play. Actually, our strategy better tolerates contention when updates appear as the contention-friendly skip list is up to 2.5× faster than the ConcurrentSkipListMap.

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 ConcurrentSkipListMap 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×.

6 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 drop. As future work, we would like to derive new contention-friendly data structures.

References

[1] S. Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749, 2007.

[2] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *PPoPP*, 2012.

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

[4] 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.

[5] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, 2009.

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

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

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

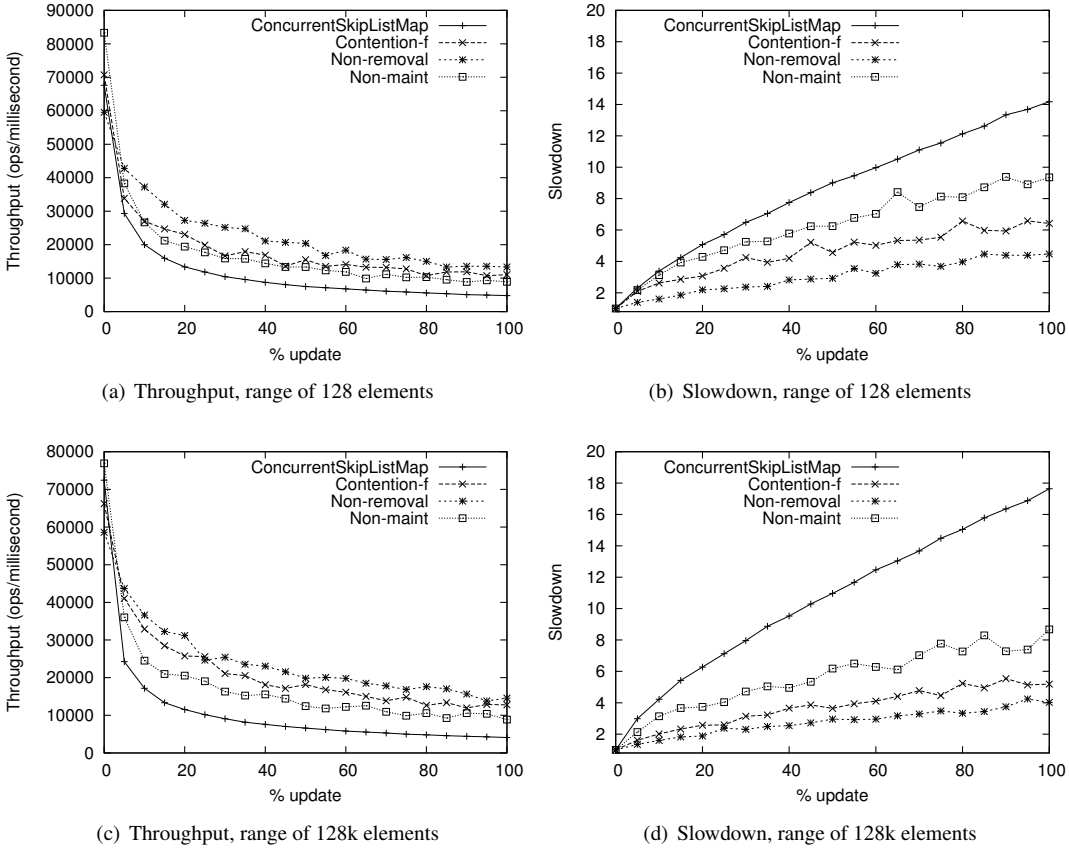
[9] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[10] D. Lea. Jsr-166 specification request group.

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

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

[13] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33, June 1990.



(a) Throughput, range of 128 elements

(b) Slowdown, range of 128 elements

(c) Throughput, range of 128k elements

(d) Slowdown, range of 128k elements

Figure 4: Comparison of % update ratio using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64 elements

[14] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

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

[16] G. Taubenfeld. Contention-sensitive data structures and algorithms. In *DISC*, pages 157–171, 2009.

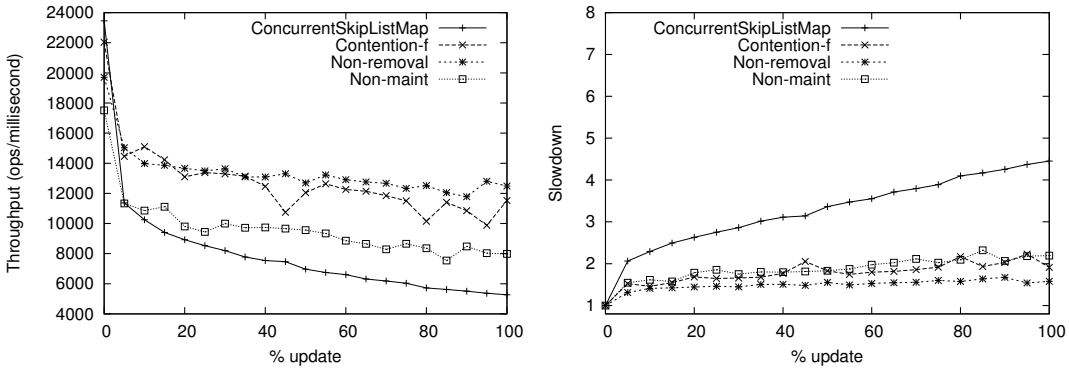
[17] J. D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, 1996.

A Additional Evaluation

In this section we present additional performance results. Each benchmark was run 4 times (the average of the 4 runs is presented in the figures) each with a duration of 5 seconds with the JVM being warmed up for 5 seconds prior to running each benchmark. In order to better understand where the benefits of the contention friendly algorithm are coming from we present two additional variations of the algorithm:

- **Non-removal contention-friendly version:** This version of the algorithm (Non-removal) does not perform any physical removals. A node that is deleted is marked as deleted, but stays in the skip list forever. This algorithm helps us examine the cost of contention caused by physical removals. A dedicated maintenance thread that takes care of modifications to the upper list levels.

- **Non-maintenance contention-friendly version:** This version of the algorithm (Non-maint) has no dedicated maintenance thread. It only uses the selective removals concept of contention friendliness; only nodes of height 1 are physically removed. This version helps us examine the benefits of using a maintenance thread. Given that there is no maintenance thread, modifications to the upper list levels are done as part of



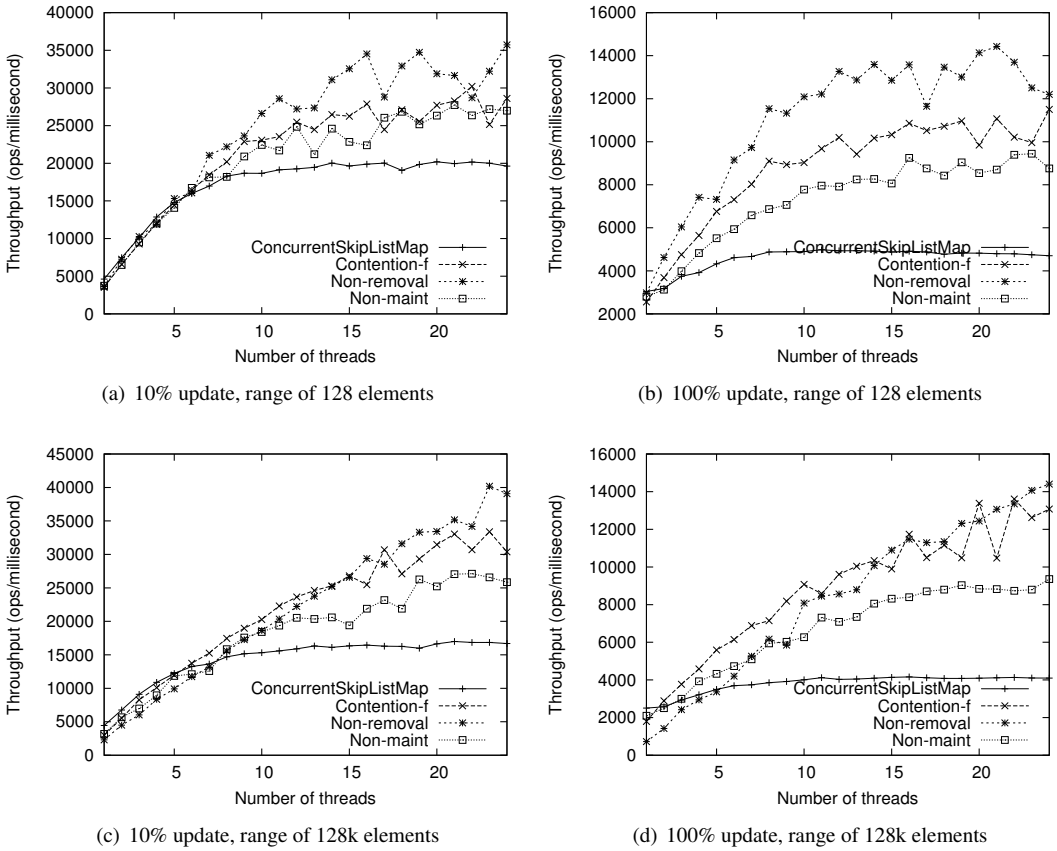
(a) Throughput, range of 128k elements

(b) Slowdown, range of 128k elements

(c) Throughput, range of 128 million elements

(d) Slowdown, range of 128 million elements

Figure 5: Comparison of % update ratio using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64k elements



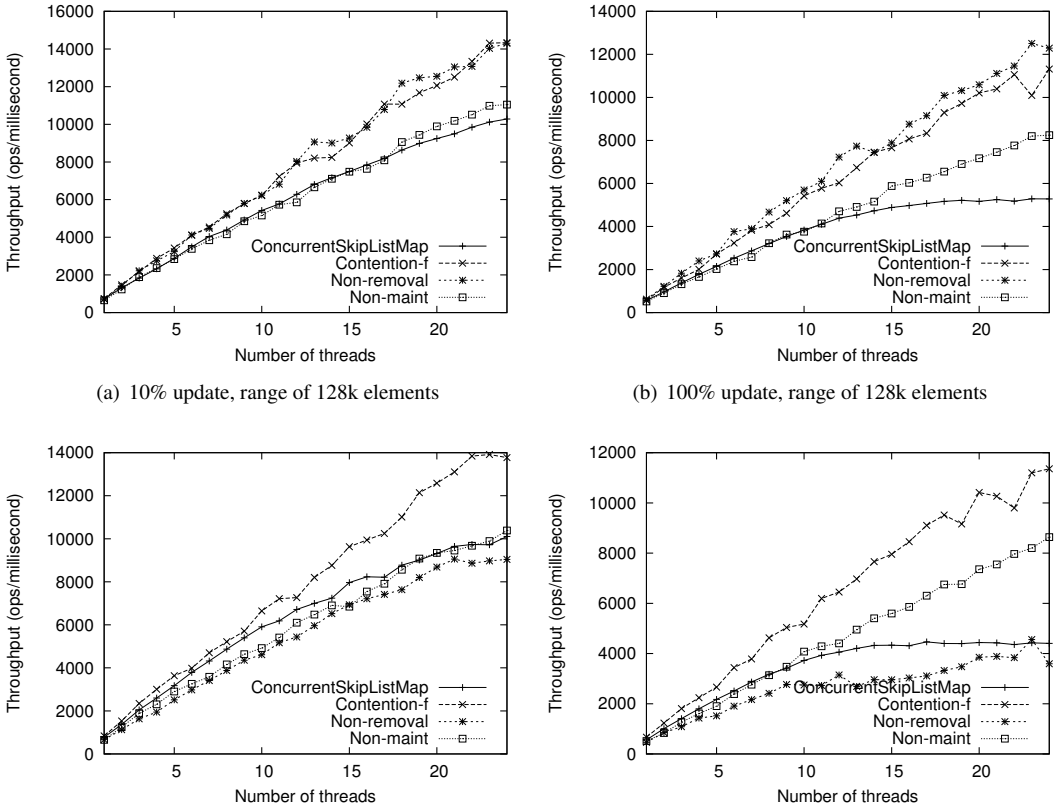
(a) 10% update, range of 128 elements

(b) 100% update, range of 128 elements

(c) 10% update, range of 128k elements

(d) 100% update, range of 128k elements

Figure 6: Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64 elements



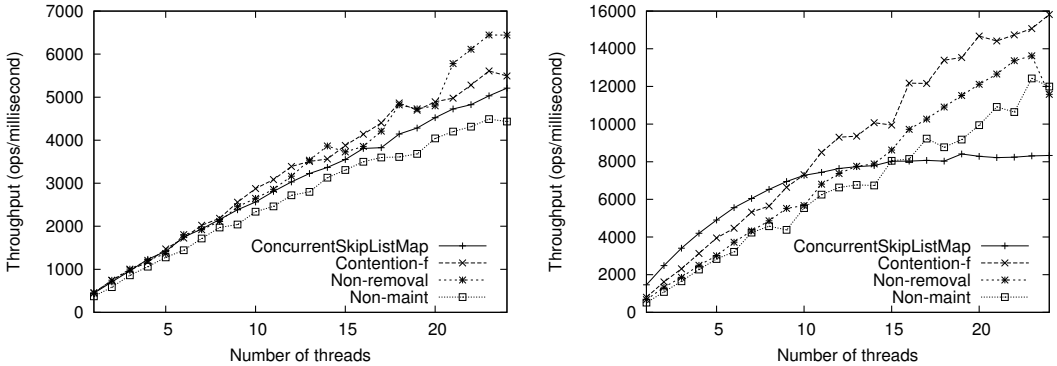
(a) 10% update, range of 128k elements

(b) 100% update, range of 128k elements

(c) 10% update, range of 128 million elements

(d) 100% update, range of 128 million elements

Figure 7: Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64k elements



(a) Grow benchmark

(b) Shrink benchmark

Figure 8: Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using the grow and shrink benchmark

the abstract operations. A node's height, like in a traditional skip list, is chosen during the *insert* operation by a random function with the one exception that a height greater than 1 will only be chosen if both node's neighbors have a height of 1. This helps prevent there from being "too many" tall nodes due to the fact that only nodes of height 1 are physically removed.

Figure 4(a)-4(d) compares the effect of increasing the amount of update operations on the algorithms. The update ratio start at 0% and is increased to 100% (50% effective). The set contains approximately 64 elements throughout the benchmarks with small variations due to concurrency. The range of elements the abstract operations can choose from is either 128 or 128 thousand. 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. The graphs show both higher performance and less slowdown of the contention friendly algorithms compared to ConcurrentSkipListMap. Between the contention-friendly versions we see that Non-removal provides the best performance. This can be explained by the fact that since the size of the set is so small performing marked removals is much less contention-effective than physically removing nodes.

Figure 5(a)-5(d) is the same benchmark as Figure 4(a)-4(d) except it is run with a set size of approximately 64 thousand elements using ranges of 128 thousand and 128 million. Contention-f shows both good performance and small slowdown while Non-maint shows performance in-between Contention-f and ConcurrentSkipListMap. Non-removal shows the best performance with the range of 128 thousand elements, but performs poorly when the range is set to 128 million elements. Since the range is so large and Non-removal does not perform any physical removals the number of marked deleted nodes in the skip list grows so large that the cost of traversal becomes more expensive than in the other algorithms. In this benchmark we see the number of nodes in the skip list to be as large as 6 million.

Figure 6(a)-6(d) tests the scalability of the algorithms by showing the effect of increasing the number of threads from 1 to 24. The tests were done with a 10% and 100% update ratios with a set size of approximately 64 elements. Here we see better scalability from the contention-friendly algorithms compared to the ConcurrentSkipListMap. Non-removal shows in general the best performance due to the reduced contention from not doing physical removals (thanks especially to the small set size), followed by Contention-f, Non-maint, and finally ConcurrentSkipListMap.

Figure 7(a)-7(d) also tests the scalability of the algorithms and is the same benchmark as 6(a)-6(d) except it is run with a set size of approximately 64 thousand elements using ranges of 128 thousand and 128 million. At 10% update all algorithms show good scalability, while at 100% updates ConcurrentSkipListMap does not scale as well, with *non-removal* scaling the worst in the larger range benchmark due to it not physically removing nodes. In general *Contention-f* is the best performer followed by Non-maint.

The purpose of figure 8(a)-8(b) is to test the scalability of the algorithms when the number of elements in the set changes by a large amount. 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. Both benchmarks are executed with a 50% update ratio. All algorithms show good scalability in the grow benchmark with a small performance advantage going to the algorithms with maintenance threads (Contention-f, Non-removal) thanks to not requiring synchronization operations to the towers. In the shrink benchmark we see that the ConcurrentSkipListMap performs best at small thread counts while the contention-friendly algorithms show better scalability. Due to the decreasing list size Contention-f calls the *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. *lower-index-level* is called by the maintenance 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 often.

B Correctness

Here we show that the CF non-blocking skip list implements a map that is linearizable [9]. The proof is separated into two parts, first we show that performing contains, insert, and delete always results in a *valid* skip list structure. Second we show that each operation has a linearization point.

Definitions. The skip list presented here represents the set (or map) abstract data type. A key k is in the set if there is a path from the field *top* to a node with key equal to k with a non-$\perp$ value, otherwise it is not in the set. Therefore a *valid* skip list has the following properties: (i) the nodes in the skip list are sorted by their keys in

ascending order, (ii) there is a path from the field *top* to at most one node with a key k at any point in time and (iii) every node with value $v \neq$ *node* has a path to it from *top*. We consider that an operation contains, insert, and delete is a *success* if it returns true, otherwise it *fails*.

For the sake of simplicity the structure is initialized with a single node with key $-\infty$ and a tower of maximum height. Each of the IndexItems of the tower have their *right* pointer initialized to $\perp$ and the node's *next* pointer is also initialized to $\perp$.

Before we define the linearization points of the operations we need the following two lemmas.

Lemma 1 *If a node n is such that n.next* $= n'$ *where* n'*.marker* $=$ true *and n.value* $= n$ *then there is no longer a path from top to n.*

Proof sketch. Nodes are only unlinked from the list during a help-remove operation. To prove the lemma it needs to be shown that the only two nodes unlinked from the list are *node* and the marker node that follows it in the list. The CAS on line 117 ensure that there are no nodes in between *node* and its predecessor. Lines 112-113 CAS exactly one marker node after *node*. To show that no nodes are added before or after the marker node we will show that pointers to and from marker nodes are never modified by considering all locations where the structure of the list is modified. First we consider the places where a new node can be added to the list. There are two places were a new node can be added to the list, this is on line 113 of the help-remove procedure and line 45 of the insert operation. In the case of the help-remove procedure, before adding a new marker node it checks that the predecessor and the successor nodes are not markers (line 106 and 109). The same is done during the insert operation on lines 87 and 92 of get-next-node and 44 of insert. The only other place where the list structure is modified is on line 117 of the help-remove procedure, but line 115 checks that the pointer modified is not from a marker node. □

Lemma 2 *A successful* remove *operation on a node of a valid skip list results in a valid skip list with the node physically removed from the list (i.e. no path from top to the node exists). The state of the elements in the abstraction is left unchanged.*

Proof sketch. The operation starts by performing a CAS on the v field of the node, changing it from $\perp$ to point to the node. Atomically changing the value from $\perp$ ensures that the key of this node was not in the set when the removal starts. If this CAS succeeds then the help-remove procedure is called. This procedure starts by ensuring a marker node is the following node in the list (line 109). If not then such a node is allocated and added to the list using a CAS (lines 112-113). The CAS ensues the pointer has not changed since it was first read on line 108 or 114 so that no newly inserted nodes are lost. The last modification done by the removal operation is the CAS done on line 117 which unlinks *node* and its successor (the marker node) from the list ensuring. To finish showing that that the removal does not modify the set it needs to be shown that only these two nodes are unlinked from the list by this CAS, but this is ensured by lemma 1. □

Lemma 3 *A* contains *operation performed on a valid skip list is linearizable and results in a valid skip list.*

Proof sketch. The contains operation does not modify the skip list so it always results in a valid list.

Success. A successful contains operation means that an element with key k exists in the set. Therefore the following must be true at its linearization point: There exists a node n with key k, $v \neq \perp$, and $v \neq n$. The linearization point for this is line 68 where the operation reads the v field. From the previous line (67) it knows that the node's key is equal to k and the checks on line 69 ensure $v \neq \perp$ and $v \neq n$.

Failure. A successful contains operation means that an element with key k does not exist in the set. There are two cases:

1. The operation finds no node with key k and returns false. This means that the check of the key on line 67 must have failed. Now the following must be true at the operation's linearization point: There does not exist a node n with key k and $v \neq n$ in the list. The linearization point is line 88 of *get-next-node* where the *next* pointer of *node* is read. First notice that the operation never traverses past a node with key larger then k (line 75 of *get-next-index* and line 92 of *get-next-node*). Therefore this *node* has a smaller key then k and must be in the list by line 87 of *get-next-node* and lemma 1. Also given that the list is valid the nodes are sorted by their keys in ascending order and that the next node in the list has a larger key then k (by line 92 of *get-next-node*) so there exists no node with key k in the list.

2. The operation finds a node in the list with key k that has been marked deleted. This means that the check of the key on line 67 must have succeeded. Now the following must be true at the operation's linearization point: There exists a node n with key k and $v = \perp$. The linearization point for this is line 87 of *get-next-node* where the node is seen to have $v \neq node$. Given that the list is valid, there exists no other node with key k in the list and since the v field of the node is neither *node* (line 87) nor not equal to $\perp$ (line 69) then it must be $\perp$ at the linearization point and k does not exist in the set.

□

Lemma 4 *An* insert *operation performed on a valid list is linearizable and results in a valid list.*

Proof sketch. **Success.** A successful insert operation means that an element with key k was added to the set. There are two cases:

1. The operation found a node with key k that was marked deleted. In this case following must be true before the linearization point: There exists a node with key k and $v = \perp$. And after the linearization point: There exists exactly one node with key k, $v \neq \perp$, $v \neq node$, and $v \neq node$. The linearization point for this is when the value of the node is CASed from $\perp$ to v on line 35. This CAS ensures the precondition because of the validation (that checks that the node's key is k and value is $\perp$) done on lines 87 and 92 of *get-next-node* and line 34 of *insert* must be valid for the CAS to succeed. This CAS (line 35) then also produces the post condition by updating the node's value to the value that was given as input. In this case no modifications are made to the structure of the list or to any nodes with $v = node$ so the resulting list must still be *valid*.

2. The operation found no node with key k in the list. In this case following must be true before the linearization point: There does not exist a node with key k, and $v \neq node$. And after the linearization point: There exists exactly one node that has a path to from *top* with key k, $v \neq \perp$, and $v \neq node$. The linearization point is when the newly allocated node is linked into the list by changing the predecessor p's *next* pointer to point to the new node (line 45 using a CAS. The following checks ensure that the node is inserted in the correct location in the list (ensuring the sorted property of the *valid* list) and that it is not inserted before or after a marker node which ensures by lemma 1 that the predecessor is in the list. Line 87 of *get-next-node* ensures that the predecessor is not a marker $p.v \neq p$, line 92 of *get-next-node* ensures that the successor has key greater than k and predecessor has key smaller than k, and line 44 of *insert* ensures that the successor is not a marker by checking that its value is not a pointer to itself. Given that a node's key never changes and that a node that is allocated as a marker always stays a marker, the CAS ensures that the predecessor and the successor are the same nodes and the checks are still valid. Within the *setup_node* operation a new node is allocated, has its key, value and pointers set (lines 59-61). This setup followed by the CAS ensures the post condition.

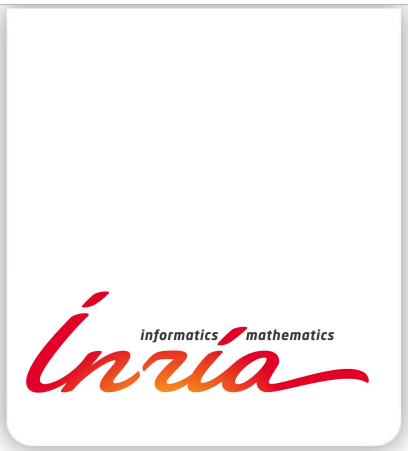
Failure. A failed insert operation follows the same structure as a successful *contains* operation.

□

Lemma 5 *A* delete *operation performed on a valid list is linearizable and results in a valid list.*

Proof sketch. **Success.** A successful delete operation means that an element with key k was removed from the the set. This means that the operation found a node with key k that was not marked deleted. In this case following must be true before the linearization point: There exists a node with key k, $v \neq \perp$, and $v \neq node$. And after the linearization point: There exists exactly one node with key k and $v = \perp$. The linearization point for this is when the value of the node is changed to $\perp$ by the CAS on line 22. This CAS ensures the precondition because of the validation (that checks that the node's key is k and value is neither $\perp$ nor *node*) done on lines 87 and 92 of *get-next-node* and line 21 of *delete* must be valid for the CAS to succeed. This CAS (line 22) also produces the post condition by changing the nodes value to $\perp$. In this case no modifications are made to the structure of the list or to any nodes with $v = node$ so the resulting list must still be *valid*.

Failure. A failed delete operation follows the same structure as a failed *contains* operation. □



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399