



HAL
open science

A Contention-Friendly, Non-Blocking Skip List

Tyler Crain, Vincent Gramoli, Michel Raynal

► **To cite this version:**

Tyler Crain, Vincent Gramoli, Michel Raynal. A Contention-Friendly, Non-Blocking Skip List. [Research Report] RR-7969, 2012. hal-00699794v3

HAL Id: hal-00699794

<https://inria.hal.science/hal-00699794v3>

Submitted on 16 Oct 2012 (v3), last revised 16 Nov 2012 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



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



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 — 20 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 $2.5\times$ 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 à saut (skip list)

Résumé : Ce rapport présente une approche méthodologique pour les structures de recherche concurrentes avec des applications aux non-bloquant listes à 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 become otherwise the bottlenecks of every day applications.

The big-oh complexity, which indicates the worst-case amount of accesses in an operation, used to prevail in the choice of a particular data structure algorithm running in a sequential context or with limited concurrency. Yet contention may become an even more important factor of performance drops in today's multicore systems. For example, some concurrent data structures are so contended that they cannot perform better than bare sequential code, and exploiting additional cores simply makes the problem worse [15].

Skip lists are increasingly popular alternatives to B-trees in main-memory databases, like *memsql*¹, as they can be traversed in sorted order. In short, a skip list is a linked structure that diminishes the linear big-oh complexity of a linked list by having nodes with additional shortcuts pointing towards other nodes located further in the list [14], hence tall nodes are referred to as *towers*. These shortcuts allow 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 machines 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 structure itself and returning as soon as possible for the sake of responsiveness.
- The *lazy selective adaptation*, which aims at being postponed, adapts the skip list structure to the abstract changes by re-arranging elements or garbage collecting deleted ones.

More specifically, the aforementioned decoupling translates into: (i) 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 (ii) splitting an element removal into a logical deletion marking phase and its physical removal and garbage collection. This adaptation is selective as only low contention towers, which are typically the shortest, are physically removed.

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

Specifically, this decoupling led us to adopt novel algorithmic designs. First, decoupling the adaptations from responsive operations lets us adapt deterministically the heights of the towers of the skip list to obtain a more balanced structure. By contrast, traditional adaptations are tightly coupled to abstract modifications and rely on pseudo-random functions to achieve reasonable efficiency. Second, in the case where a large amount of short towers have been removed (leaving too many tall towers), we decrease their heights by simply removing the bottom level of each tower rather than modifying their frequently accessed topmost level. Third, this decoupling allows us to centralize all adaptation tasks on a single thread, meaning that only accesses to the bottom most level needs to be synchronized.

Additionally, our contention-friendly skip list is *non-blocking*, ensuring that the system as a whole always makes progress. In particular, no thread ever blocks waiting for some resources. This property guarantees that one slow thread does not affect the overall system performance, which represents 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 is to crash, it neither makes the skip list inconsistent nor does it prevent other threads from accessing resources. More precisely, 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. In particular, we compare our algorithm's performance against the Java adaptation by Lea of Harris, Michael and

¹<http://developers.memsql.com/docs/1b/indexes.html>

Fraser’s algorithms [8, 9, 12]. 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 up to a $2.5\times$ 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 6 presents the experimental results and Section 7 concludes.

2 Related Work

Decoupling each data structure modification into multiple tasks has proved beneficial for memory management [5] and efficiency [3, 13], yet this idea was essentially applied to balanced trees but not to diminish contention in skip lists.

The work in this paper was motivated by [3] in which similar ideas were applied to a tree data structure in transactional memory, but here the ideas are modified and new difficulties are realized in order to deal with the intricacies of creating highly efficient and concurrent lock-free code.

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 mostly 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 [12] 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 Java [11]. 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 height is determined by a pseudo-random function such that the probability for a tower to have height ℓ is $2^{-O(\ell)}$. A tower of height 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 [19] 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 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 delete nodes they encounter to avoid traversing superfluous towers. This approach differs from the former one where a superfluous tower is traversed by the lookup, marking all its nodes as deleted. Fomitchev and Ruppert also use the logical deletion mechanism when 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.

In contrast with previous algorithms, our non-blocking skip list decouples the abstract modification from the selective structural adaptation to achieve contention-friendliness. In particular, the physical removal applies selectively to towers of low heights 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 modifying nodes at the bottom level.

Finally, transactional memories can be used to implement non-blocking skip list, however, they may restrict skip list concurrency [8] or block [6]. Note that our notion of contention-friendliness differs from Gadi Taubenfeld’s contention-sensitivity that was applied to queues [18] as the latter aims at executing an efficient path before switching to a lock-based one when contention raises.

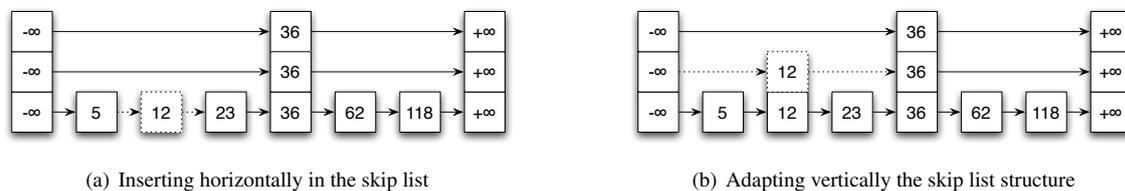


Figure 1: Decoupling the eager abstraction insertion from the lazy selective adaptation

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 [10].

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`.

3.1 Eager abstract modification

Previous skip lists generally maintain the tower per height distribution so that the probability of a tower i to have height ℓ is $\Pr[\text{height}_i = \ell] = 2^{-O(\ell)}$, hence each time the abstraction is updated, the invariant is checked and the structure is accordingly adapted as part of the single operation. 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 contention-friendly skip list only the bottom level is modified and the additional structural modification is postponed until later. In the section 5 we show that this abstract modification is sufficient to guarantee linearizability. 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, 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 tower 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 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 tower 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 tower heights.

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 insertions to execute faster: Other than the obvious fact that an insert will return without modifying the upper levels, 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 node as deleted without modifying the actual structure. Importantly, the subsequent structural adaptation selects for removal the nodes whose removal would induce the least contention, not all nodes are immediately physically removed. Only if the node only exists on the bottom level then the operation attempts to remove the node immediately (note that it may fail due to concurrent operations and be performed later by a subsequent structural adaptation).

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 short tower, 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 tower with value 36 requires updating at least 3 pointers while tower with value 62 requires updating $O(1)$ pointers. 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 62 is unlikely to be accessed by a many 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 adaptation thread. 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 of the adaptation to multiple threads. The correctness proof is deferred to the technical report.

Algorithm 1 depicts the algorithm of the eager abstract operations while Algorithm 2 depicts the algorithm of the lazy selective adaptation. The skip list algorithm is *non-blocking* meaning that 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, 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 linked lists this node has, a *marker* flag indicating whether or not the node is a marker node (used during removals).

We use the logical deletion technique [9] 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 $n.v = n$). This technique is used in order to ensure the safety of the non-blocking protocol, the details of which are discussed in Section 4.2.

The upper levels are made up of singly linked lists of items called *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 *index-items* 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 level of the skip list.

The list is initialized with a dummy node with a tower of the maximum starting height. The pointer *top* points to the first element of the highest index of the list, all traversals start from this pointer. Figure 4.3 shows the structure of the Contention-friendly skip list where the second node is in the process of being removed and the third 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 a list level 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 then 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 90) then the help-remove procedure (this call is necessary to ensure the non-blocking property of the algorithm) is called (line 91) 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 24-26) 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 43) then a CAS is performed to try marking the node as deleted (line 44). If the marking is successful then the remove operation is called, attempting to remove the node if it has a height of 1, the removal is described in section 4.2.

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 70) by unmarking it (lines 71). If no node with key k is found then the insert operation allocates “physically” a new node (line 78) before adding it to the list by performing a CAS operation on the next pointer of the predecessor (line 81). Note that existing skip list algorithms cannot exploit logical insertions as each logical deletion is traditionally followed by a physical removal. Following this the *prev* pointer of the successor of the new node is changed to point to the new node (line 82). An interesting point is that no synchronization is needed for this operation, even if this operation is delayed and a concurrent modification to the *prev* point happens, it will always point to a node with a smaller key which is all that is needed for safety.

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 task is executed after a successful delete operation and continuously by the adapting thread. The remove operation is more difficult than the abstract delete operation 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 [19]. 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 101). 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 113-114), this is similar to the process done in 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, during the insert operation, before the actual node insertion is performed a validation is executed in order to ensure that neither the predecessor nor the successor node is marked. (Lines 88 and 93 of the get-next-node procedure and line 80 of the insert operation.) To complete the removal a CAS is performed on the predecessor’s next pointer (line 118) removing both the node and its marker node from the list.

In the second task of the adapting thread the upper levels of nodes 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. Existing algorithms call a random function to calculate the heights of towers, here heights are computed deterministically. Note that this differs from traditional skip list algorithms that favor probabilistic balancing as it was more efficient in sequential executions.

This deterministic rebalancing is done using the procedure raise-index. This procedure starts by setting up an array called *first* which contains the first element of each list of index-items. The raise-index-level procedure is then 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 154–156), it increments the height of the middle node by 1 (line 163). 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 148–153). Note that the existence

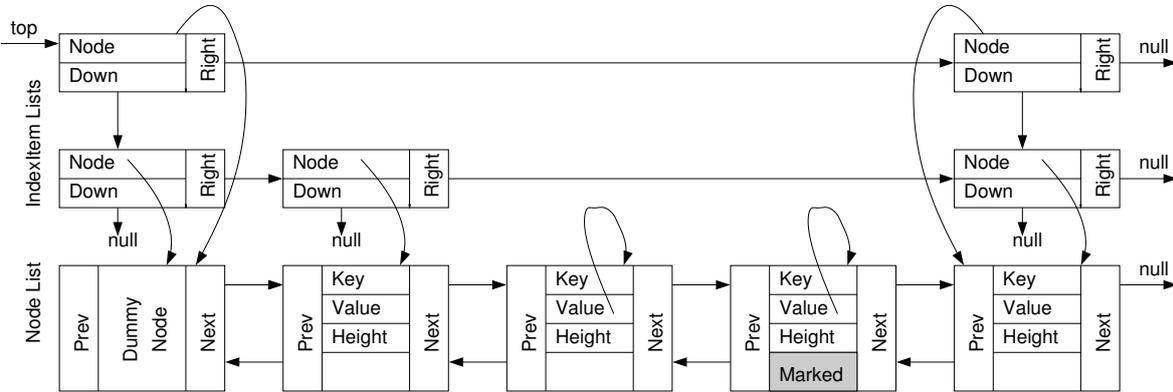


Figure 2: Skip list structure

of such items do not affect the correctness of the algorithm as search operations will either traverse past such items, or pass through them into the physically removed node resulting in the same situation as would be if the search had reached the node during traversal of the bottom list level.

If there are enough nodes in the highest index level, a new index level may be needed (i.e. the last call of `raise-index-level` returned true), this is done by the maintenance thread by simply adding a new index node to the top of the dummy node’s tower and modifying the `top` pointer (lines 137–140 of `raise-index`).

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 towers are marked as deleted while 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, removing the entire bottom *index-item* level of the skip list by changing the *down* pointers of the level above to \perp (line 125). 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 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 no matter the concurrency pattern of concurrent threads. This is similar to deadlock freedom in a crash-free blocking environment. Since none of the operations use locks or blocking operations the algorithm can be proven to be non-blocking by showing that any thread performing an abstract operation can be in an infinite loop if and only if there some other threads that are concurrently making progress on other abstract operations. We will now look at each loop in the program, showing that they can only loop infinitely in this case.

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 the technical report 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.

This is true for the while loop in the `get-next-index` procedure. Iterations of this loop traverses forward and down levels of *index-items* and will only loop infinitely if there are an infinite number of concurrent insertions.

The `help-remove` procedure has a while loop on lines 110–115, 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 88–88) and an outer level loop that traverses forward in the list (lines 87–96). An infinite traversal in the inner loop would require an infinite number of node removals which would also require an 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 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 successful concurrent 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 71 of insert and line 44 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 81 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 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].

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 second 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. 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 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 thirds responsible for physical deletions.

5 Correctness

Here we show that the CF non-blocking skip list implements a map that is linearizable [10]. 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 $v \neq \text{node}$ 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 $n.\text{next} = n'$ where $n'.\text{marker} = \text{true}$ and $n.\text{value} = n$ 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 118 ensure that there are no nodes in between *node* and its predecessor. Lines 113-114 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 where a new node can be added to the list, this is on line 114 of the help-remove procedure and line 81 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 107 and 110). The same is done during the insert operation on lines 88 and 93 of get-next-node and 80 of insert. The only other place where the list structure is modified is on line 118 of the help-remove procedure, but line 116 checks that the pointer modified is not from a marker node. \square

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 110). If not then such a node is allocated and added to the list using a CAS (lines 113-114). The CAS ensures the pointer has not changed since it was first read on line 109 or 115 so that no newly inserted nodes are lost. The last modification done by the removal operation is the CAS done on line 118 which unlinks *node* and its successor (the marker node) from the list ensuring. To finish showing 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. \square

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 25 where the operation reads the v field. From the previous line (24) it knows that the node's key is equal to k and the checks on line 26 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 24 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 89 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 than k (line 55 of *get-next-index* and line 93 of *get-next-node*). Therefore this *node* has a smaller key than k and must be in the list by line 88 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 than k (by line 93 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 24 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 88 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 88) nor not equal to \perp (line 26) 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 71. 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 88 and 93 of *get-next-node* and line 70 of *insert* must be valid for the CAS to succeed. This CAS (line 71) 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 81 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 88 of *get-next-node* ensures that the predecessor is not a marker $p.v \neq p$, line 93 of *get-next-node* ensures that the successor has key greater than k and predecessor has key smaller than k , and line 80 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 30-32). 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 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 44. 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 88 and 93 of *get-next-node* and line 43 of *delete* must be valid for the CAS to succeed. This CAS (line 44) 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. □

6 Evaluation

Here we compare our skip list to the `java.util.concurrent` skip list on a multi-core machine.

Settings. 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. 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 modifies 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 [8, 9, 12]. 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.

6.1 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. 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 $2.5\times$ 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 $1.8\times$.

6.2 On the effect of maintenance and removal

In this section we present additional performance results in order to better assess the the performance benefits of the contention-friendly skip list. Experiments are run using the same setup as in the previous section. 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 the methodology.

- **Non-removal contention-friendly version:** This version of the algorithm (Non-removal) 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 modifications to the upper list levels.

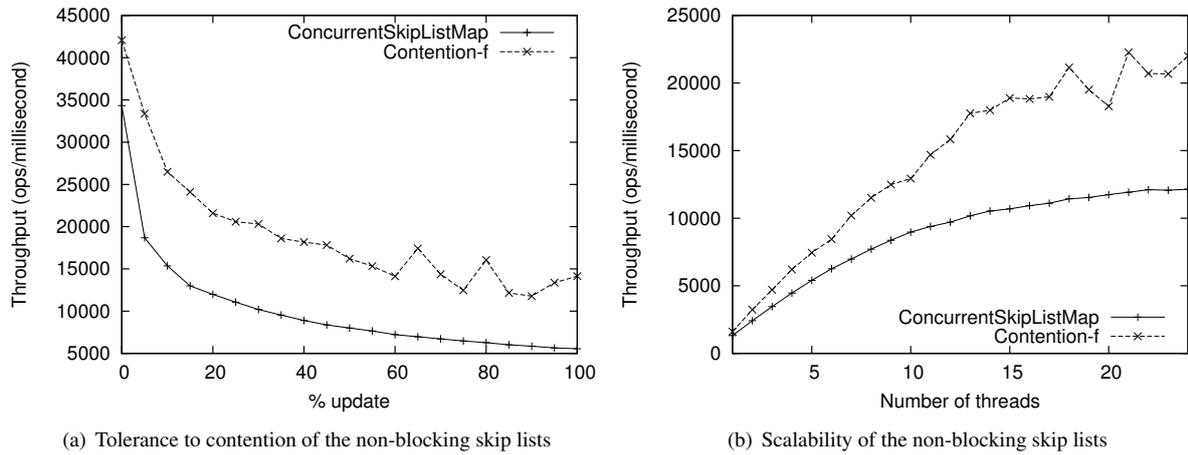


Figure 3: Comparison of our contention-friendly non-blocking skip list against the JDK concurrent skip list (ConcurrentSkipListMap)

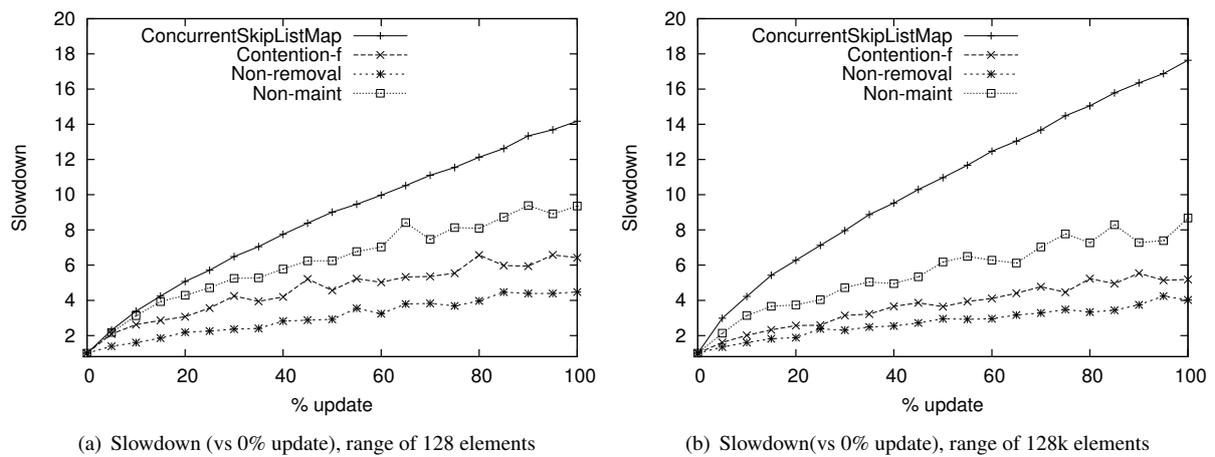


Figure 4: Comparison of % update ratio using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64 elements (24 threads)

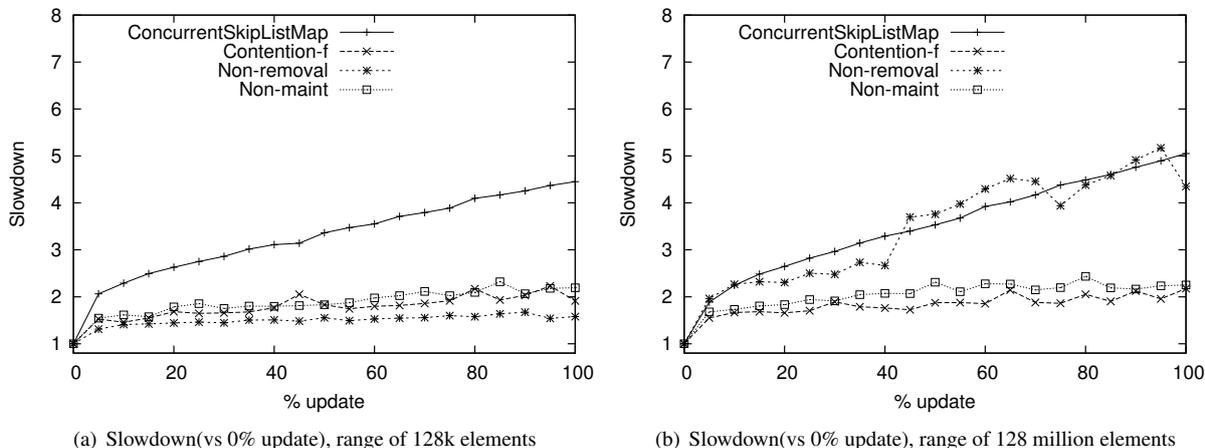


Figure 5: Comparison of % update ratio using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64k elements (24 threads)

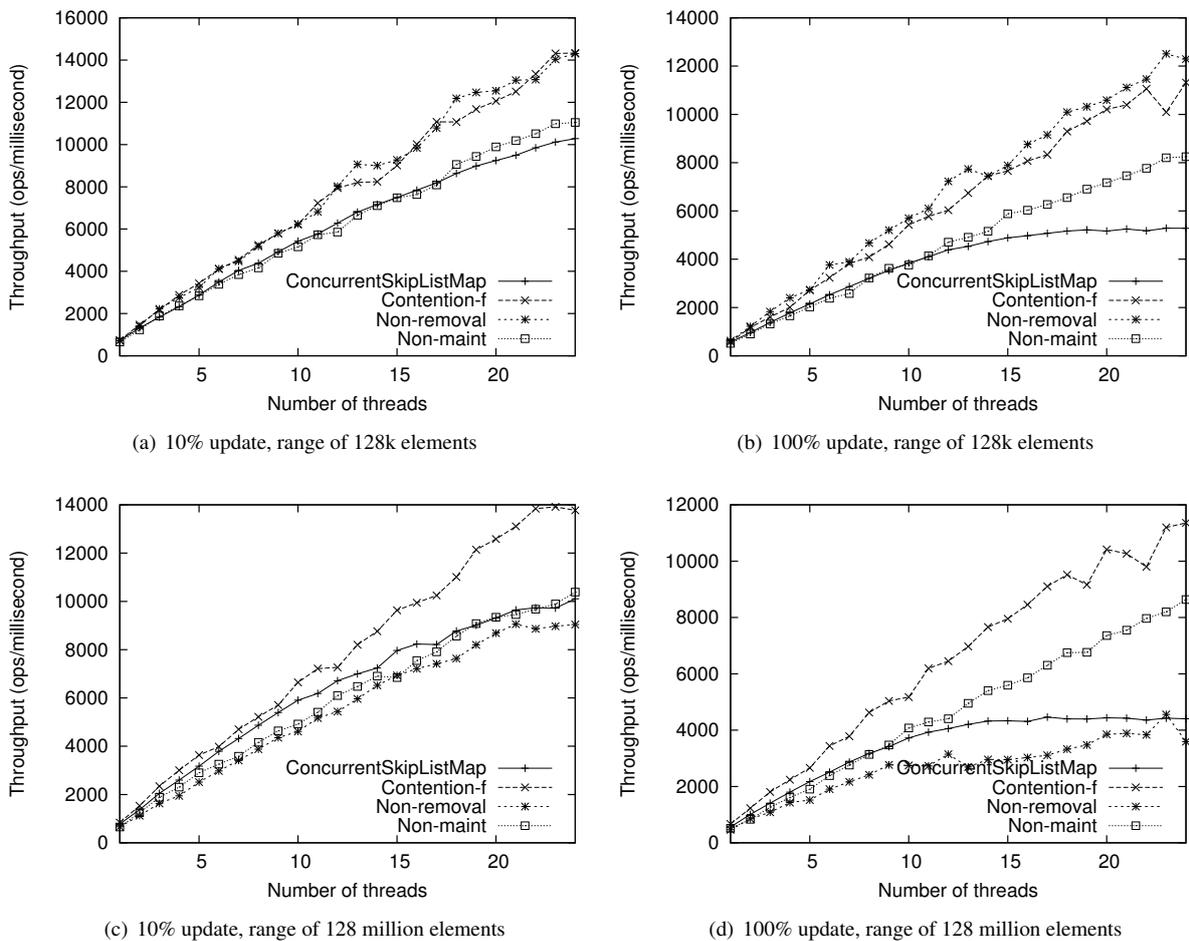


Figure 6: Comparison of increasing number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using a set of size 64k elements

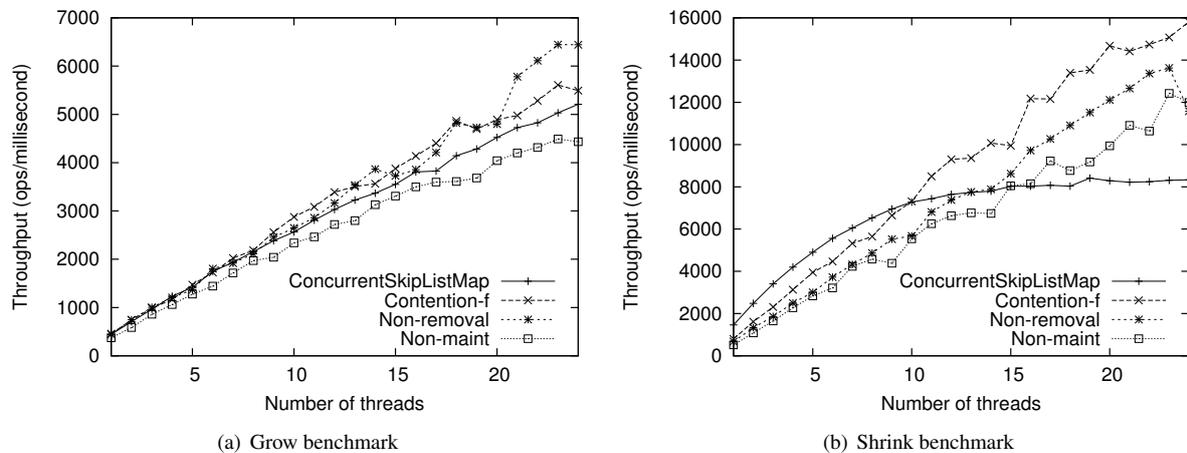


Figure 7: Comparison of number of threads using our non-blocking skip lists against the JDK concurrent skip list (ConcurrentSkipListMap) using the grow and shrink benchmark (24 threads)

- Non-maintenance contention-friendly version:** This version of the algorithm (Non-maint) has no dedicated adapting 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 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(b) compares the effect of increasing the amount of update operations on the algorithms showing the slowdown compared to a 0% update ratio. These experiments were executed using 24 threads. The update ratio starts at 0% and is increased to 100% (50% effective). The set is very small, containing 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 (figure 4(a)) or 128 thousand (figure 4(b)). 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. For both ranges the slowdown is much greater for the ConcurrentSkipListMap compared to any of the other algorithms. For the ConcurrentSkipListMap we see up to nearly an $18\times$ slowdown at 100% update with the large range while the worst any of the others is less than $10\times$. Between these 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. Contention-f performs well also, showing a maximum slowdown of around $4\times$, with Non-maint performing in-between Contention-f and ConcurrentSkipListMap.

Figure 5(a)-5(b) is the same benchmark as Figure 4(a)-4(b) except it is run with a much larger set size of approximately 64 thousand elements using ranges of 128 thousand (figure 5(a)) and 128 million (figure 5(b)). 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. Given that the range is very 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.

6.3 Evaluating scalability

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 using ranges of 128 thousand and 128 million. At 10% update all algorithms show good scalability for both ranges.

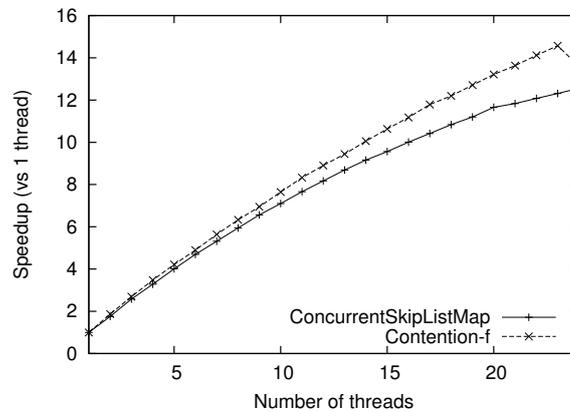


Figure 8: Speed-up of the Contention-Friendly (CF) and non-CF data structures using a modified version of the SPECjbb 2005 benchmark

Things get more interesting at 100% update. For either range `ConcurrentSkipListMap` starts losing scalability at around 12 cores, while `Contention-f` performs well all the way up to 24-cores. `Non-maint` scales well, but does not perform quite as well as `Contention-f`, showing the advantage of having the maintenance thread. *Non-removal* 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.

6.4 Quality of the structural adaptation

The purpose of figures 7(a)-7(b) is to test whether or not the maintenance thread is able to do its job when the number of elements in the set change 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 using 24 threads. Both benchmarks are executed with a 50% update ratio.

In the grow benchmark (figure 7(a)) all algorithms show good scalability with a small performance advantage going to the algorithms with maintenance threads (`Contention-f`, `Non-removal`). This is likely due to the fact that these algorithms do not requiring synchronization operations on the towers.

In the shrink benchmark (figure 7(b)) 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 or less often, resulting in a skip list with a larger or smaller amount of marked deleted nodes.

6.5 SPECjbb 2005

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. These collections implement the Java Map interface so they can be swapped for the `ConcurrentSkipListMap` or the contention-friendly 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 [2] 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 are only once piece of a larger program. 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 scalability results of running the modified benchmark from 1 to 24 threads are shown in figure 8. The first thing that should be noticed is that both skip list implementations show very good scalability. At 23 cores the `ConcurrentSkipListMap` skip list benchmark shows a $12.3\times$ speed-up over a single core, while the contention-friendly skip list benchmark shows a $14.6\times$ speed-up. There is obviously very little contention on the concurrent map, likely comparable to 1% or 2% update ratio of the micro-benchmark as at 5% we are already seeing large slowdowns (figures 4(a)–4(b) 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 contention-friendly skip list as the core count approaches 23. At 23 cores the `ConcurrentSkipListMap` version performs 747730 business operations per second (bops), the instrument of measure for SPECjbb performance, while the contention-friendly 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 `ConcurrentSkipListMap` version has a small, but significant (7000 bops) performance advantage over the contention-friendly one which diminishes until 19 cores when the contention-friendly skip list takes over in performance. We believe this to be due to the fact that the `ConcurrentSkipListMap` 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 contention-friendly version takes a large performance hit when going from 23 to 24 cores because at that point the adapting thread and program threads must compete for processor time. Such observations tend to indicate that the contention-friendly skip list is really beneficial when there are less application threads running than available cores.

7 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] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2007.
- [3] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *PPoPP*, 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*, 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*, pages 300–314, 2001.
- [10] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.
- [11] D. Lea. Jsr-166 specification request group.
- [12] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

- [13] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *PODS*, 1987.
- [14] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33, June 1990.
- [15] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.
- [16] Standard performance evaluation corporation, specjbb2005 benchmark, 2005.
- [17] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *SAC*, pages 1438–1445, 2004.
- [18] G. Taubenfeld. Contention-sensitive data structures and algorithms. In *DISC*, pages 157–171, 2009.
- [19] J. D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, 1996.

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:  $marker \in \{\text{true}, \text{false}\}$ , 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:  $contains(k)_p$ :
21:  $item \leftarrow \text{traverse-tower}(top, k)$  // find tower
22:  $node \leftarrow \text{get-next-node}(item, k)$  // find node
23:  $result \leftarrow \text{false}$ 
24: if  $node.k = k$  then
25:    $v \leftarrow node.v$ 
26:   if  $(v \neq \perp \wedge v \neq node)$  then
27:      $result \leftarrow \text{true}$ 
28:   return  $result$ 

29:  $setup\text{-}node(node, k, v)_p$ :
30:  $new.k \leftarrow k$ ;  $new.v \leftarrow v$  // allocate node called new
31:  $new.prev \leftarrow node$ 
32:  $new.next \leftarrow node.next$ 
33: return  $new$ 

34:  $delete(k)_p$ :
35:  $node \leftarrow \text{traverse-tower}(top, k)$ 
36: while true do
37:    $node \leftarrow \text{get-next-node}(node, k)$ 
38:   if  $node.k \neq k$  then
39:      $result \leftarrow \text{false}$ 
40:     break()
41:   else
42:      $v \leftarrow node.v$ 
43:     if  $(v \neq \perp \wedge v \neq node)$  then
44:       if  $\text{CAS}(node.v, v, \perp)$  then
45:          $\text{remove}(node.prev, node)$ 
46:          $result \leftarrow \text{true}$ 
47:         break()
48:     else
49:        $result \leftarrow \text{false}$ 
50:       break()
51:   return  $result$ 

52:  $\text{traverse-tower}(item, k)_p$ :
53: while true do
54:    $nitem \leftarrow item.right$ 
55:   if  $nitem.node.k > k$  then
56:      $nitem \leftarrow item.down$ 
57:     if  $nitem = \perp$  then
58:        $result \leftarrow item.node$ 
59:       break()
60:     else if  $nitem.node.k = k$  then
61:        $result \leftarrow item.node$ 
62:       break()
63:      $item \leftarrow nitem$ 
64:   return  $result$ 

65:  $\text{insert}(k, v)_p$ :
66:  $node \leftarrow \text{traverse-tower}(top, k)$ 
67: while true do
68:    $node \leftarrow \text{get-next-node}(node, k)$ 
69:   if  $node.k = k$  then
70:     if  $node.v = \perp$  then // logical insertion
71:       if  $\text{CAS}(node.v, \perp, v)$  then
72:          $result \leftarrow \text{true}$ 
73:         break()
74:     else
75:        $result \leftarrow \text{false}$ 
76:       break()
77:   else
78:      $new \leftarrow \text{setup-node}(node, k, v)$ 
79:      $next \leftarrow new.next$ 
80:     if  $next.val \neq next$  then // logical insertion
81:       if  $\text{CAS}(node.next, next, new)$  then
82:          $next.prev \leftarrow new$  // safe
83:          $result \leftarrow \text{true}$ 
84:         break()
85:   return  $result$ 

86:  $\text{get-next-node}(node, k)_p$ :
87: while true do
88:   while  $node.v = node$  do  $node \leftarrow node.prev$ 
89:    $next \leftarrow node.next$ 
90:   if  $(next \neq \perp \wedge next.v = next)$  then
91:      $\text{help-remove}(node, next)$  // help the removal
92:      $next \leftarrow node.next$ 
93:   if  $(next = \perp \vee next.k > k)$  then
94:      $result \leftarrow node$ 
95:     break()
96:    $node \leftarrow next$ 
97:   return  $result$ 

```

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

```

98: remove( $pred, node$ ) $_p$ :
99:    $result \leftarrow false$ 
100:  if  $node.level = 0$  then
101:     $CAS(node.v, \perp, node)$  // compare-and-swap
102:    if  $node.v = node$  then
103:       $help\_remove(pred, node)$ 
104:       $result \leftarrow true$ 
105:  return  $result$ 

106: help-remove( $pred, node$ ) $_p$ :
107:  if ( $node.val \neq node \vee node.marker$ ) then
108:    return
109:   $n \leftarrow node.next$ 
110:  while  $\neg n.marker$  do
111:     $new \leftarrow setup\_node(node, \perp, \perp)$ 
112:     $new.v \leftarrow new$ 
113:     $new.marker \leftarrow true$ 
114:     $CAS(node.next, n, new)$  // compare-and-swap
115:     $n \leftarrow node.next$ 
116:  if ( $pred.next \neq node \vee pred.marker$ ) then
117:    return
118:   $CAS(pred.next, node, n.next)$  // compare-and-swap

119: lower-index-level( $\cdot$ ) $_p$ :
120:   $index \leftarrow top$ 
121:  while  $index.down.down \neq \perp$  do
122:     $index \leftarrow index.down$ 
123:  while  $index \neq \perp$  do
124:     $index.down \leftarrow \perp$ 
125:     $index.node.height \leftarrow index.node.height - 1$ 
126:     $index \leftarrow index.next$ 
127:  // Update the index of the first array

128: raise-index( $\cdot$ ) $_p$ :
129:   $max \leftarrow 0$ 
130:   $next \leftarrow top$ 
131:  while  $next \neq \perp$  do
132:     $first[max] \leftarrow next$ 
133:     $next \leftarrow next.down$ 
134:     $max \leftarrow max + 1$ 
135:  for  $i \leftarrow max; i > 0; i \leftarrow i - 1$  do
136:     $increase-level \leftarrow raise-index-level(first[i], first[i - 1], max - i)$ 
137:  if  $increase-level$  then
138:    // allocate index-item called new
139:     $new.down \leftarrow top$ 
140:     $top \leftarrow new$ 

141: raise-index-level( $prev, prev-tall, height$ ) $_p$ :
142:   $raised \leftarrow false$ 
143:   $index \leftarrow prev.right$ 
144:  while true do
145:     $next \leftarrow index.right$ 
146:    if  $next = \perp$  then
147:      break()
148:    while  $index.node.v = index.node$  do
149:       $prev.right \leftarrow next$ 
150:      if  $next = \perp$  then
151:        break()
152:       $index \leftarrow next$ 
153:       $next \leftarrow next.right$ 
154:    if ( $prev.node.level \leq height$ 
155:       $\wedge index.node.level \leq height$ 
156:       $\wedge next.node.level \leq height$ ) then
157:       $raised \leftarrow true$ 
158:      // Allocate a new index-item called new
159:       $new.down \leftarrow index$ 
160:       $new.node \leftarrow index.node$ 
161:       $new.right \leftarrow prev-tall.right$ 
162:       $prev-tall.right \leftarrow new$ 
163:       $index.node.level \leftarrow height + 1$ 
164:       $prev-tall \leftarrow new$ 
165:       $prev \leftarrow index$ 
166:       $index \leftarrow index.right$ 
167:  return  $raised$ 

```



**RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

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

ISSN 0249-6399