A Contention-Friendly Methodology for Search Structures

Tyler Crain, Vincent Gramoli, Michel Raynal

A Contention-Friendly Methodology for Search Structures

Tyler Crain, Vincent Gramoli, Michel Raynal
tyler.crain@irisa.fr, vincent.gramoli@epfl.ch, raynal@irisa.fr

RESEARCH REPORT
N° 1989
Febuary 2012
Project-Team ASAP

ISSN 0249-6399 ISRN INRIA/RR--1989--FR+ENG

A Contention-Friendly Methodology for Search Structures

Tyler Crain[*], Vincent Gramoli[†], Michel Raynal[‡*]
tyler.crain@irisa.fr, vincent.gramoli@epfl.ch, raynal@irisa.fr

Project-Team ASAP

Research Report n° 1989 — Febuary 2012 — 32 pages

Abstract: In this paper, a new methodology for writing concurrent data structures is proposed. This methodology limits the high contention induced by today's mutlicore environments to come up with efficient alternatives to most widely used search structures, including skip lists, binary search trees and hash tables.

Data structures are generally constrained to guarantee a big-oh step complexity even in the presence of concurrency. By contrast our methodology guarantees 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 access* that returns rapidly for efficiency reason and a *lazy structural adaptation* that may be postponed to diminish contention.

We illustrate our methodology with three contention-friendly data structures: a lock based skip list and binary search tree, and a lock-free hash table. Our evaluation clearly shows that our contention-friendly data structures are more efficient than their non-contention-friendly counterparts. In particular, our lock-based skip list is up to 1.3× faster than the Java concurrent skip list, our lock-based tree is up to 2.2× faster than the most recent concurrent tree algorithm we are aware of, and our lock-free hash table outperforms by up to 1.2× the Java concurrent hash table. We also present contention-friendly versions of the skip list and binary search tree using transactional memory. Even though our transaction-based data structures are substantially slower than our lock-based ones, they inherit compositionality from transactional memory and outperform their non-contention-friendly counterparts by 1.5× on average.

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

[*] IRISA, Université de Rennes 35042 Rennes Cedex, France
[†] EPFL
[‡] Institut Universitaire de France

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Une approche méthodologique pour l'implémentation efficace de structures de recherche concurrentes

Résumé : Ce rapport présente une approche méthodologique pour les structures de recherche concurrentes avec des applcations aux listes á saut (skip list), arbres et table de hachage (hash table).

Mots-clés : mémoire transactionnelle, arbre binaire, 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 usually called *manycore* [2]. Programmers must thus change their habits to design new concurrent data structures that can be bottlenecks in 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 [30]. In response to such contention, researchers seek relaxed abstractions, i.e., alternative abstractions offering weaker guarantees, whose performance remains acceptable when their data structure implementation is placed in a highly concurrent context. This is typically the case for the queue of the Intel® TBB[1] that is not FIFO under multiple producers/consumers and for the quiescently consistent stack that is not LIFO in the presence of concurrency [30].

To better illustrate how contention can counterbalance the big-oh complexity in today's multi-/many-cores, Figure 1 depicts the performance of a 48-core machine running the same set based experiment on a concurrent linked list, with $O(n)$ complexity, and on a concurrent skip list, with $O(\log_2 n)$ complexity. A skip list, in short, is a structure that diminishes the complexity of a linked list by being a sort of linked list whose nodes may have additional shortcuts pointing towards other nodes located further in the list [28]. In this experiment, 48 threads run insert/delete/contains accesses with an increasing proportion of update accesses over read-only ones on each of these two structures initialized with 512 elements.[2] To obtain the corresponding concurrent data structures used in the experiments, we simply encapsulated the sequential code of each access into an elastic transaction [9]. Interestingly, above 20% updates the concurrent linked list is more efficient than the concurrent skip list—this is shown by the negative values of the speedup-1. The reason is that the linked list updates are localized, that is, each of them only af-

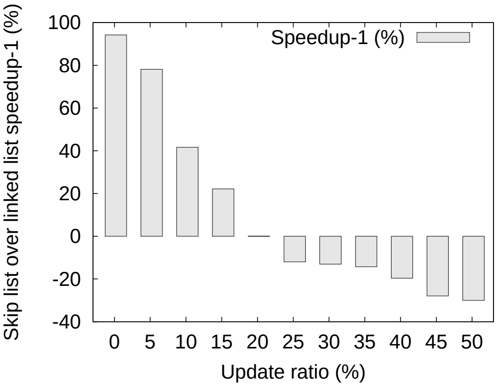


Figure 1: Impact of contention on the performance of two 512-sized data structures (with 48 cores running with an increasing update ratio)

[1] Intel® Threading Building Blocks (TBB) http://threadingbuildingblocks.org.

[2] More precisely, this experiment was performed on a 4 × 12-core AMD Opteron machine running at 2.1GHz and 32 GB of memory, each point is averaged over 5 runs of 5 seconds each, removes and inserts accesses are triggered with the same probability to keep the size expectation constant and removes/inserts that do not update the data structure are considered read-only accesses.

fects a constant number of nodes, typically the predecessor of the removed or of the newly inserted node. By contrast, the skip list updates may affect up to a logarithmic amount of predecessors for each removed of newly inserted node, producing additional contention.

This result is unsurprising as Herb Sutter noticed that linked list could better tolerate contention than balanced trees for similar reasons [31], yet it is interesting to observe experimentally that only 20% updates make a linear complexity data structure better suited than a logarithmic complexity data structure on nowadays' multicore machines.

In the light of the impact of contention on performance, we propose the *Contention-Friendly (CF)* methodology as a methodology to design new data structures that accommodate contention of modern multi-/many-core machines without relaxing the correctness of the abstractions. To this end, we argue for a genuine decoupling of each access into an eager abstract access and a lazy structural adaptation. The abstract access consists in modifying the abstraction by minimizing the impact on the structure itself and aims at returning as soon as possible for the sake of responsiveness. The structural adaptation, which can be deferred until later, aims at adapting the structure to these changes by re-arranging elements or garbage collecting deleted ones.

We illustrate the CF methodology by designing three data structures with locks, universal primitives, and transactions: a skip list, a binary search tree and a hash table. As for the skip list, the aforementioned decoupling translates into splitting a node insertion into the insertion phase at the bottom level of a skip list and the structural adaptation responsible for updating pointers at its higher levels, or into splitting a node removal into a logical deletion marking phase and its physical removal and garbage collection. Similarly, the decoupling of the binary tree accesses consists in inserting or logically removing a node prior to rebalancing and/or garbage collecting. Finally, the hash table decoupling lies in inserting/deleting eagerly and resizing the structure lazily.

Our Java implementation of the resulting data structures indicates that our methodology leads to good performance on today's multicore machines. In particular using a micro benchmark, on a 64-way Niagara 2 machine our lock-based CF binary search tree improves the performance of the most recent Java lock-based binary search tree implementation [4] by up to 2.2×, our lock-based CF skip list improves the performance of Doug Lea's concurrent skip list adaptation of Harris and Michael algorithms [14,23] by up to 1.3×, and our lock-free hash table outperforms by up to 1.2× the JDK hash table, which is widely distributed in the `java.util.concurrent` package. Finally, we show that state-of-the-art software transactional memories execute 1.5× faster on average when the data structures are contention-friendly.

Section 2 describes the related work. Section 3 depicts the CF methodology, Section 4 illustrates it on three data structures. Section 5 presents the experimental results and Section 6 concludes. The companion appendix comprises the pseudo-code and correctness proofs of our CF algorithms, as well as additional experimentations using transaction-based variants of our CF algorithms and a discussion.

2 Related Work

Various complexity metrics exist to evaluate data structures efficiency on a given workloads. From a theoretical point of view, the big-oh notation helps to derive data structures whose access step complexity is proportional to the total number of elements. Typically, balanced trees have a logarithmic big-oh access complexity whereas nonoverloaded hash tables have a constant big-oh access complexity. This big-oh complexity does not capture the cost of contention and theoretical models have been explored to remedy this issue [7]. Unfortunately, there are not enough evaluations of this impact in practice.

From a more pragmatic point of view, the locality of data both in terms of space (i.e., the promiscuity of data stored in memory) and time (i.e., the closeness of the points in time at which they are accesses) has been an important metric of consideration when implementing data structures in cache-coherent systems. Cache-aware and cache-oblivious data structures try to exploit locality to maximize the chance of cache hits. While the former data structures rely on some tunable parameter that can accommodate the targeted platform like the Judy array[3], the later aims at being more portable by flattening cleverly structural nodes into memory [12], for example the tree algorithm of van Emde Boas et al. [32]. Both approaches are tied to cache-coherent machines but do not accommodate upcoming many-core platforms whose cache-coherence is either limited [33] or absent [22].

The decoupling of the update and rebalancing was vastly explored in the context of trees [1, 3, 5, 13, 19, 21, 26, 27] but this idea was not generalized to other search structures. The decoupling of the removals in logical and physical phases was originally studied in transactional systems [24] and later applied to various lock-free data structures including linked lists [14], hash tables [23], skip lists [10, 11] and binary search trees [8] but insertions in these data structures were not decoupled. The contention friendly methodology generalizes these decoupling into an eager abstract access and a lazy structural adaptation that benefit both insertions and removals.

Our methodology is independent from the synchronization primitive used but lies essentially in splitting accesses into an eager abstract access and a lazy structural adaptation. Although we focus essentially on lock-based data structures, we also evaluate the benefit of various transactional memory algorithms when running our contention-friendly data structures. We have already illustrated the benefit of decoupling accesses into separate transactions in [5] on a C-based binary search tree. In such optimistic executions, this decoupling translated into avoiding a conflict with a rotation from rolling back the preceding insertion/removal. Here we generalize our previous work by showing how a similar decoupling can benefit pessimistic execution and various search structures and we compare our results to existing Java concurrent structures. Previous investigations on improving the performance of transaction-based data structures focused exclusively on the improvement of the transaction algorithm. Some of these investigations led to the development of novel transaction models based on abstract locks to ignore low level conflicts [15, 25], or elastic transactions [9].

Finally, Shavit suggests to relax data structure guarantees in the light of the new multicore context [30]. A stack algorithm and several relaxations to this algorithm are presented to support concurrency. The objective as well as the means to achieve it are quite different from ours. First, the problem raised by placing the stack into the multicore context is that performance drops below the sequential stack performance, and the goal is to diminish contention to limit this concurrency drawback. By contrast, we focus on deriving alternative data structures that are more scalable than highly concurrent ones, hence leveraging multi-/many-cores. Second, the goal of limiting contention induced by multiple cores is achieved by relaxing consistency. In the stack example, this relaxation boils down to replacing linearizability by quiescent consistency, guaranteeing that the last-in-first-out policy of an access is only with respect to preceding

[3] http://judy.sourceforge.net

Data structure	**Invariant**	**Abstract modifications**	**Structural adaptations**
Hash tables	constant load factor (i.e., $\#\mathsf{nodes}/\#\mathsf{buckets} = O(1)$)	key-value pair insertion logical deletion	adding buckets and rehashing physical deletion + rehashing
Search trees	balance (i.e., shortest route to leaf $\approxeq$ longest route to leaf)	node insertion logical deletion	rotation physical deletion + rotation
Skip lists	node distribution per level (i.e., $\Pr[level_i = j] = 2^{O(j)}$)	horizontal insertion logical deletion	vertical insertion + increasing toplevel physical removal + decreasing toplevel

Table 1: Decoupling example of existing data structure accesses into an abstract modification and a structural adaptation

calls when no other accesses execute concurrently. Conversely, the contention friendly methodology aims at replacing existing data structures without relaxing their abstraction consistency: all accesses remain linearizable.

3 The CF Methodology at a Glance

In this section, we give an overview of the Contention-Friendly (CF) methodology by describing how to write contention-friendly data structures.

The CF methodology aims at modifying the implementation of existing data structures using two simple rules without relaxing their correctness. The correctness criterion ensured here is linearizability [18]. The data structures considered are *search structures* because they organize a set of items referred to as *elements* in a way that allows to retrieve the unique expected position of an element given its value. The typical abstraction implemented by such structures is a collection of elements that can be specialized into various sub-abstractions like a set (without duplicates) or a dictionary (that maps each element to some value). We consider *insert*, *delete* and *contains* operations that respectively inserts a new node associated to a given value, removes the node associated to a given value or leaves the structure unchanged if no such node is present, and returns the node associated to a given value or $\perp$ if such a node is absent. Both inserts and deletes are considered *updates*, even though they may not modify the structure.

The key rule of the methodology is to decouple each update into an *eager abstract modification* and a *lazy structural adaptation*. The secondary rule is to make the removal of nodes selective and tentatively affect the less loaded nodes of the data structure. These rules induce slight changes to the original data structures as summarized in Table 1, that result in a corresponding data structure that we denote using the *contention-friendly* adjective to differentiate them from their original counterpart.

3.1 Eager abstract modification

Existing search structures rely on strict invariants (cf. Table 1) to guarantee their big-oh complexity, hence each time the structure gets updated, the invariant is checked and the structure is accordingly adapted instantaneously. While the update may affect a small sub-part of the abstraction, its associated restructuring is a global modification that conflict potentially with any concurrent update, thus increasing contention.

The CF methodology aims at minimizing such contention by returning eagerly the modifications of the update operation that makes the changes to the abstraction visible. By returning eagerly, each individual process can move on to the next operation prior to adapting the structure. 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 such complexity may not be the predominant factor in contended execution as we reported in the Introduction.

A second advantage is that removing the structural adaption from the abstract modification makes the cost of each operation more predictable. All operations share similar cost and create the same amount of contention. More importantly the completion of the abstract operation does not depend on the structural adaptation (like they do in existing algorithms) so the structural adaptation can be performed differently, using and depending on global information.

The skip list example. A traditional skip list picks a level for each node when they are inserted based on some pseudo-random function. The aim of this function is to distribute the levels so that operations have an average cost of $O(\log n)$. In certain workloads this can be preferred over trees due to the assumption that rotations are more costly. When a node is inserted in the contention-friendly skip list it has a level of one, which is all that is needed to ensure the correctness of the abstraction.

As an example, assume we aim at inserting an element with value 12 in a skip list. Our insertion consists in an abstract modification that updates only the bottom most level by inserting the new node as if its level was the lowest one leading to Figure 2 where dashed arrows indicate the freshly modified pointers. We defer the process of linking this same node at higher levels, to diminish the probability of having this insertion conflict with a traversing operation.

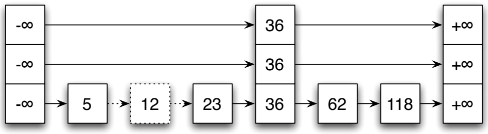


Figure 2: Inserting horizontally in the skip list

3.2 Lazy structural adaptation

The purpose of decoupling the structural adaptation from the preceding abstract modification is to enable its postponing (by, for example, dedicating a separate thread to this task), hence the term "lazy" structural adaptation. The main intuition here is that this structural adaptation is intend to ensure the big-oh complexity rater than to ensure correctness of the state of the abstraction. Hence, the linearization point belongs to the execution of the abstract modification and not the structural adaptation and postponing the structural adaptation does not change the effectiveness of operations. The visible modification applied to the abstraction (and the structure) during the abstract modification guarantees that any further operation applying to the same structure will observe the changes. This helps ensuring that all operations are linearizable in that real-time precedence is satisfied. In Appendix C we show that our structures implement a linearizable abstraction.

This postponing has several advantages whose prominent one is to enable merging of multiple adaptations in one simplified step. Although the structural adaptation might be executed in a distributed fashion, by each individual updater threads, one can consider centralizing it at one dedicated thread. Since these data structures are designed for architectures that use many cores performing the structural adaptation on a dedicated single separate thread, takes advantage of hardware that might otherwise be left idle. Only one adaptation might be necessary for several abstract modifications and minimizing the number of adaptations decreases accordingly the induced contention. Furthermore, several adaptations can compensate each other as two restructuring can lead to identity. For example, a left rotation executing before a right rotation at the same node may lead back to the initial state and executing the left rotation lazily makes it possible to identify that executing these rotations is useless.

The skip list example. As explained in the previous example the insertion executes in two steps. Once the horizontal insertion of node 12, depicted in Figure 2, is complete, a restructuring is necessary to ensure the logarithmic complexity of further accesses. A separate structural adaptation step is accordingly raised to increase the node level appropriately. The insertion at higher levels of the skip list is executed as a separate step, which guarantees eventually a good distribution of nodes among levels as depicted in Figure 3. This decoupling allows higher concurrency by splitting one atomic operation into two atomic operations.

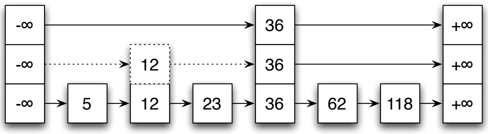


Figure 3: Adapting vertically the skip list structure

3.3 Selective removal

In addition to decoupling level adjustments, we do selective removals. A node that is deleted is not removed instantaneously, instead it is marked as deleted. The structural adaptation then selects cleverly nodes that are suitable for removal, i.e., whose removal would not induce high contention. This is important because removals may be expensive. Removing a frequently accessed node requires locking or invalidating a larger portion of the structure. Removing such a node is likely to cause much more contention than removing a less frequently accessed one. In order to prevent this, only nodes that are marked as deleted and have a level of 1 (in the skip list) or a single or no children (in the tree) are removed. This leads to less contention, but also means that certain nodes that are marked as deleted will not be removed. In the tree it has already been observed that only removing such nodes [5], [4] results in a similar sized structure as existing algorithms. In the skip list the level of a node is calculated in such a way that after a structural adaptation is performed less than half the nodes (in the worst case) in the list will be marked as deleted. In practice this number is observed to be much smaller.

The skip list example. Let us look at a specific example with the skip list. On the one hand, a removal of a node with a high level, say the one with value 36 in Figure 3, 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 ℓ pointers which increase the probability of conflict with a concurrent operation accessing the same pointers, hence removing node with value 36 requires to update 3 pointers while node with value 63 requires to update a single pointer. Second, the organization of the skip list implies that higher level pointers are more likely accessed by any operation, hence the removal of 36 typically conflicts with every operation concurrently traversing this structure (because all these operations would follow the topmost left pointer) whereas the single next pointer of 62 is unlikely accessed by concurrent traversals. Removing a tall node such as 36 would also mean that in order to keep the logarithmic complexity of the traversals a node would have to take its place at an equivalent height.

3.4 Avoiding contention during traversal

Each abstract operation (*contains*, *insert*, *delete*) of a tree or a skip list is expected to traverse $O(\log n)$ nodes. Given that the traversal is the longest part of the operation, the CF algorithms try to avoid as often as possible producing contention. Concurrent data structures often require more complex synchronization operations during traversal (not including the updates done after the traversal). For example, locking nodes in a tree helps ensure that the traversal remains on track during a concurrent rotation [4], using compare-and-swap operations during traversal helps the raising and lowering of levels of a concurrent *insert*/*delete* in a lock-free skip list [11], or using optimistic strategy helps at the risk of having to restart [16, 17].

Usually these synchronization operations are required due to structural adaptations and the CF algorithms structural adapt differently to especially so that operations can avoid using locks or synchronization operations during traversal.

4 Putting the CF Methodology to Work

Here we present how we apply the contention-friendly (CF) methodology to three data structures. For further detail on the algorithms and correctness proofs please refer to Appendix B and Appendix C, respectively.

4.1 CF Skip list

The CF skip list is made up of several levels of linked lists, with the bottom level being a doubly linked list. Each node on the bottom level contains the following fields: A key *k*, a *next* and *prev* node pointers, a lock field, a *del* flag indicating if the node has been marked deleted, and a *rem* flag indicating if the node has been physically removed. The algorithm presented here is lock-based, however, we have derived a transaction-based version (cf. Appendix A).

Abstract operations. The goal of these CF algorithms is for the abstract operations to encounter and produce as little contention as possible. In particular, it boils down to setting the nodes *del* flag to true to delete a node as well as linking a new node to the bottom list level to insert it. These modifications are necessary to guarantee that linearizability, with all other structural adaptations being saved for later execution. For the sake of safety, the abstract insertion acquires a lock on the predecessor node of the to-be-inserted node whereas the abstract deletion acquires a lock on the to-be-marked node. The lock is immediately released after the insertion or deletion completes.

No locks are acquired during the traversal, inducing no contention. More precisely, while traversing upper levels the operation will move forward in the list using the *next* pointer until it encounters a node with a larger key than the one being search for at which point it will move down a level, similarly to a bare sequential implementation would do. At the bottom level the traversal may end up on a node that is physically removed due to a concurrent structural adaptation *remove* operation, in this case it travels backwards in the list following the *prev* pointer until it arrives at a node that has not yet been removed.

Structural adaptation. The first task of the structural adaptation is to remove nodes marked as deleted who have a height of 1. In order to prevent conflicts with concurrent abstract operations the node n to be removed and its predecessor in the list ($n.prev$) are locked. The prior nodes *next* pointer ($n.prev.next$) is then modified so that it points to the next node ($n.next$), and the next node's *prev* pointer ($n.next.prev$) is then modified to point to the previous node ($n.prev$). Finally the n's *rem* flag is set to true and the locks are released.

The structural adaptation must also modify the level 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, but if this same function was used here the structure would end up with excessive tall nodes.

When choosing the heights it is important to consider that the fundamental structure of a skip list is not designed to be perfectly balanced but rather probabilistically balanced. Consider a perfectly balanced skip list. The node in the very middle of the list would be the tallest node and the nodes just to the right and left of this node would be nodes with height 1. Now if a couple new nodes are inserted at the very end of the list then to re-balance the skip list the node that was previously the tallest node would now be shrunk to a level of 1, and one of its neighboring nodes which previously had height of 1 would become the tallest node. Instead a scheme of approximately balanced is more fitting for the skip list (as this is what the existing algorithm's random functions do).

By contrast, the CF skip list deterministically adjusts the level of nodes. From the bottom level going upwards, it traverses the entire list of the level, and each time it observes that 3 consecutive nodes whose height equals this level, it raises the level of the second of this node (the one in the middle) by 1. Such a technique approximates the targeted number of nodes present at each level, balancing the structure. Doing this is similar to the original intuition of the skip list, there is no frequent re-balancing going on, tall nodes will stay tall nodes. Less modification of the taller nodes also means less contention at the frequently traversed locations of the structure.

Given that the number of nodes in the list might also shrink the height of nodes might also be lowered. When the height of the tallest node is greater than some threshold (usually when the height is greater than the log of the total number of nodes in the list) the entire bottom index level of the skip list is simply removed by modifying the *down* pointers of the level above. Doing this avoids constant modification of the taller nodes and ensures there are not too many marked deleted nodes left in the list.

4.2 CF Tree

The CF tree is a binary search tree. Each of its nodes contains the following fields: a key k, pointers l and r to the left and right children nodes, a lock field, a *del* flag indicating if the node has been marked deleted, and a *rem* flag indicating if the node has been physically removed. As for the CF skip list, the CF tree algorithm presented here is lock-based but we also derived a transaction-based variant of it.

Abstract operations. Similarly to the CF skip list operations the *insert* and *delete* operations must acquire a lock on the node they modify. A *delete* operation sets the node's *del* flag to `true` while an *insert* operation allocates a new node and modifies the parent's child pointer to point to it.

The traversal is performed without locks. At each node the traversal travels to the right child if the node's key is larger than k, otherwise it travels to the left child. Since locks are not used, the traversal might get caught during a concurrent removal or rotation, but the structural adaptation is done in such a way that the traversal can continue safely following the child pointers.

Structural adaptation. The structural adaptation is in charge of removing marked deleted nodes that have at most one non-$\perp$ child pointer. Removals are done by first locking the node n to be deleted and its parent. The parent's child pointer is then modified so that it points to n's non-$\perp$ child (if any). Next n's child pointers are modified so that they point upwards to it's parent node allowing concurrent traversal that arrived on this node a safe path back to the tree. Finally n's *rem* flag is set to true and the locks are released.

The structural adaptation must also perform rotations in order to ensure the tree is balanced so that traversal can be done in $O(\log n)$ time. Methods for performing localized rotation operations in the binary trees have already been examined and proposed in several works such as [4,5]. The main concept used here is to propagate the balance information from a leaf to the root. A leaf is known to have height of 0 for their left and right children. This information is then propagated upwards by sending the height of the child to the parent where the value is then increased by 1. Local rotations are performed depending on this information and result eventually in a balanced tree.

In order to avoid using locks and aborts/rollbacks during traversals, rotations are performed differently than traditional rotations. Before performing the rotation the parent node and its child node that will be rotated are locked in order to prevent conflicts with concurrent *insert* and *delete* operations. In a traditional rotation there is one node n that is rotated downwards and one node (one of n's children) that is rotated upwards. A traversal preempted on the node rotated downwards (n in this case) is then in danger of being set off track and missing the node it is searching for. The rotations performed in the CF algorithm avoid this by not actually rotating n at all, meaning that after the rotation n still has a pointer to the node that is rotated upwards allowing traversals to continue safely. Instead a new node takes n place in the structure. This new node is set to have the same values and pointers as n would if a rotation was performed as normal. After the rotation, the node n has its *rem* flag set to true and, finally, the locks are released.

4.3 CF Hash table

The CF hash table contains an array of pointers with each location pointing to $\perp$ or to a list of nodes. Each node contains the following fields. A key k, and a *next* pointer pointing to the next node in the list. This algorithm is lock-free (relying on compare-and-swap for synchronization) but we derived a transaction-based variant of it (cf. Appendix A).

Abstract operations. Given that the traversal for the *contains*, *insert*, *delete* operations has complexity $O(1)$ and not $O(\log n)$ the hash table operations are performed slightly differently. In fact, the shortness of the hash table operations brings two main differences to the algorithm.

First physical removals are done from within the *delete* operation. This is because the contention caused by removing the node will only be with other nodes of that bucket which are expected to be $O(1)$.

Second the algorithm is made lock-free because given the short operations, a cache miss caused by loading a lock could be relatively costly. Other implementations might avoid this by using coarser grained locks, like lock-striping, but this can cause contention on the lock(s). Instead we use a lock-free implementation where each operation only uses (at most) a single synchronization operation, which is a compare-and-swap on the given bucket pointer.

For the sake of linearizability of operations the compare-and-swap always happens at the same location (on the bucket pointer) and the *next* pointer of list elements is never modified after node creation. An *insert* will compare-and-swap a new node as the first element of the list, while a *delete* will remove a node by creating a new list that does not contain the node and compare-and-swap this new list to the bucket. If the compare-and-swap fails due to a concurrent operation then the operation retries from the beginning.

Structural adaptation. The structural adaptation must ensure the $O(1)$ cost of *contains*, *insert*, *delete* operations. This is done by rehashing and resizing the table which first traverses the table counting the number of nodes. If the number of nodes is greater than some threshold (usually a fraction of the number of buckets in the table) then a rehash is performed and the size of the table is increased by a size of the power of 2.

The rehash is performed one bucket at a time allowing concurrent operations on other buckets. At each bucket the list of nodes is copied and placed into two new lists added to the corresponding buckets of the new table. Next a compare-and-swap is performed at the bucket of the old table replacing the list there with a dummy node. If the compare-and-swap fails then the rehash operation is retried for this bucket. Any abstract operation that encounters a dummy node then knows that the bucket has been rehashed so it uses the new table for the operation.

5 Evaluation

We evaluate the CF methodology using a micro benchmark by comparing our CF data structures to three Java state-of-the-art concurrent data structure implementations:

- Non-CF hash table: the widely deployed ConcurrentHashMap of the java.util.concurrent package,
- Non-CF binary tree: the most recent lock-based binary search tree [4] we are aware of, and
- Non-CF skip list: the Doug Lea's ConcurrentSkipListMap relying on Harris and Michael algorithms [14, 23].

All CF data structure implementations use a separate thread in addition to the application threads that constantly adapts the structure to compensate the effect of preceding abstract modifications. We use an UltraSPARC T2 with 8 cores running up to 8 hardware threads each, comprising 64 hardware threads in total. For each run we averaged the number of executed operations per microsecond over 5 runs of 5 seconds. Thread counts are $1,2,4,8,16,24,32,40,48,56$ and 64 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.

Figure 4 depicts the tolerance to contention of the various data structures. More precisely, it indicates the slowdown of each data structure under contention as the normalized ratio of its performance with non-null update ratios over its performance without updates. The slowdown of non-CF tree and skip list always more significant than the one of their CF counterpart, indicating that the CF is more tolerant to contention.

Interestingly, the slowdown of the CF hash table is higher than the one of the non-CF hash table at low levels of contention but becomes similar at high contention levels. As shown later, our CF hash table is actually very efficient on read-only workload whereas the ConcurrentHashMap relies on lock-stripes whose segments have to be loaded even on read-only workloads. This explains why CF performance drops as soon as contention appears, however, the CF hash table tolerates the contention increase as the slowdown remains almost constant, as opposed to the non-CF hash table. We played with the number of segments and we observed better scalability with more segments but lower read-only overhead with a single one. We chose 64 segments which makes threads fetch multiple segments from memory before finding them in their cache. Another advantage of using the CF hash table is not having to worry about such segments.

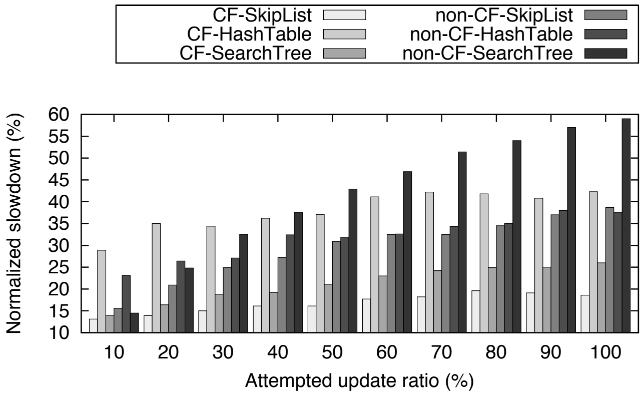


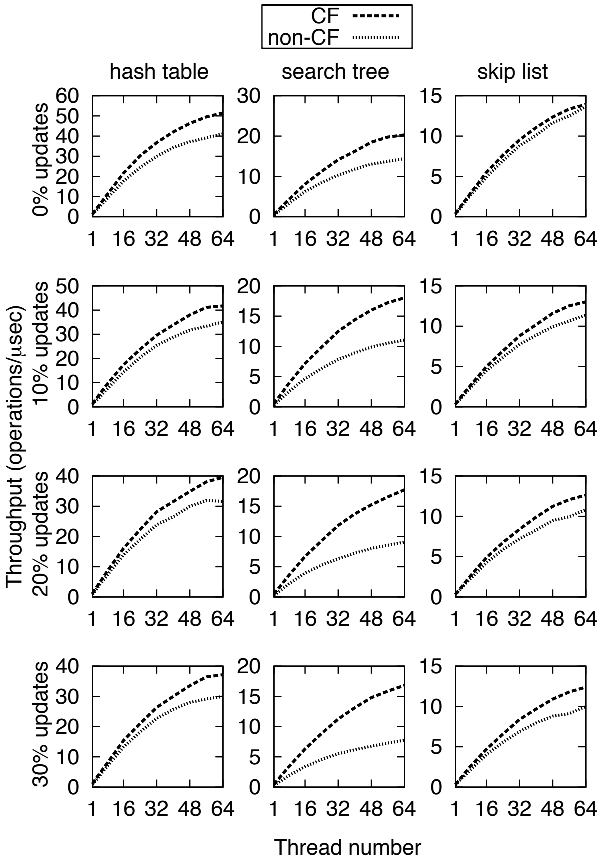
Figure 4: Tolerance to contention of Contention-Friendly (CF) and non-CF data structures (performance slowdown with respect to 0% updates)

Figure 5 compares the performance of state-of-the-art data structures against performance of our CF data structures with 2^{14} (left) and 2^{16} elements (right) and on a read-only workload (top) and workloads comprising up to 30% updates (bottom). While all data structures scale well with the number of threads, the state-of-the-art data structures are slower than their contention-friendly counterparts in all the various settings. In particular, the CF hash table, skip list, search tree are respectively up to 1.2×, 1.3×, 2.2× faster than their non-CF counterparts.

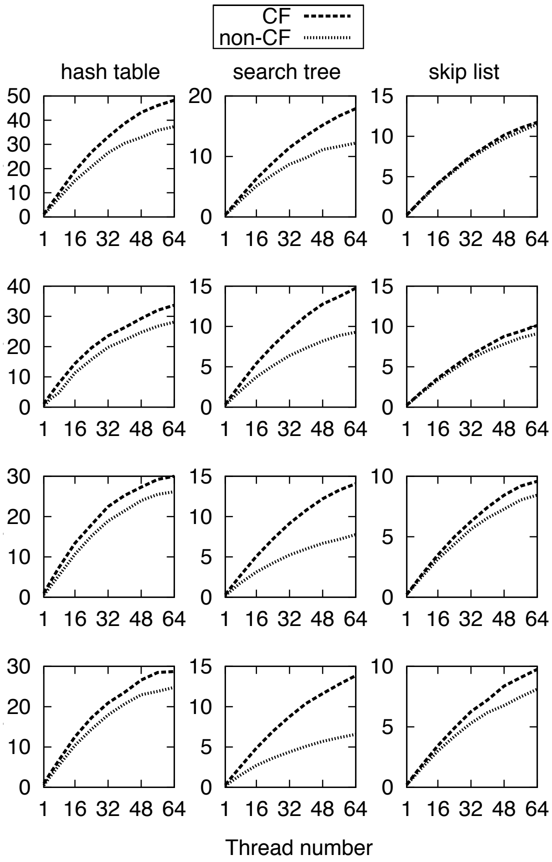
Finally Appendix A shows that our adaptation of these data structures to three transactional memory algorithms allows a performance benefit of 1.5× on average.

6 Conclusion

Multicore programming brings new challenges, like contention, that programmers have to anticipate when developing novel applications. Programmers must now give up concentrating on the big-oh complexity and should rather think in terms of contention overhead. We explored the methodology of designing contention-friendly data structures, keeping in mind that contention will be a predominant cause of performance loss



(a) 2^{14} elements



(b) 2^{16} elements

Figure 5: Performance of the Contention-Friendly (CF) and non-CF data structures

in tomorrow's architectures. This simple methodology led to a novel Java package of concurrent data structures more efficient than the best implementations we could find. We plan to extend it with additional contention-friendly data structures.

References

[1] L. Ballard. Conflict avoidance: Data structures in transactional memory, May 2006. Undergraduate thesis, Brown University.

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

[3] L. Bougé, J. Gabarro, X. Messeguer, and N. Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical Report RR1998-18, ENS Lyon, 1998.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2010.

[5] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2012.

[6] D. Dice, O. Shalev, , and N. Shavit. Transactional locking II. In *Proc. of the 20th Int'l Symp. on Distributed Computing*, 2006.

[7] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *J. ACM*, 44:779–805, November 1997.

[8] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[9] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proc. of the 23rd Int'l Symp. on Distributed Computing*, 2009.

[10] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM.

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

[12] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285 –297, 1999.

[13] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of the 19th Annual Symp. on Foundations of Computer Science*, 1978.

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

[15] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Par. Prog.*, 2008.

[16] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th international conference on Structural information and communication complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag.

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

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

[19] J. L. W. Kessels. On-the-fly optimization of data structures. *Comm. ACM*, 26:895–901, 1983.

[20] G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive software transactional memory. *Transactions on HiPEAC*, 5(2), 2010.

[21] U. Manbar and R. E. Ladner. Concurrency control in a dynamic search structure. *ACM Trans. Database Syst.*, 9(3):439–455, 1984.

[22] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. In *SC*, pages 1–11, 2010.

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

[24] C. Mohan. Commit-LSN: a novel and simple method for reducing locking and latching in transaction processing systems. In *Proc. of the 16th Int'l Conference on Very Large Data Bases*, 1990.

[25] Y. Ni, V. Menon, A.-R. Abd-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2007.

[26] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proc. of the 10th ACM Symp. on Principles of Database Systems*, 1991.

[27] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proc. of the 6th ACM Symp. on Principles of Database Systems*, 1987.

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

[29] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, 2006.

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

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

[32] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10:99–127, 1976. 10.1007/BF01683268.

[33] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

A Transactional Contention-Friendly Algorithms

The concept of splitting the abstract operation and the structural modification to create contention friendly data structures does not only apply to lock based or lock-free implementations. It can also be applied to data structures implementations using transactional memory. Our previous work has studied this problem when specifically looking at trees [5].

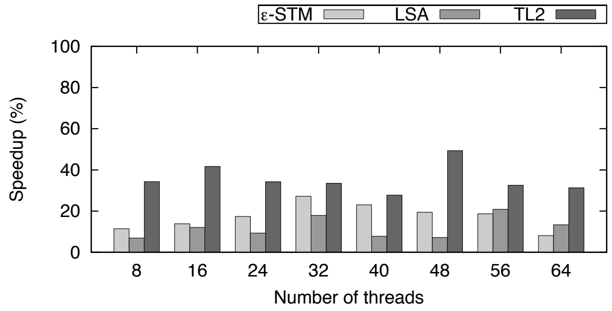
We now present the experimental results with three existing transactional memory implementations: $\mathscr{E}$-STM [9], LSA [29] and TL2 [6] using the Deuce Java bytecode instrumentation framework [20]. The experimental settings are the same as for other experiments, except that we evaluate the red-black tree (non-CF tree) and the Pugh skip list (non-CF skip list) to compare our CF tree and CF skip list against on 2^{12} elements with 5% effective updates. Figure 6(a) and Figure 6(b) depict the speedup of using the CF skip list over using the non-CF skip list (resp. CF tree over using the non-CF tree) for each of the considered transactional memory implementations. Using transaction-based CF data structures as opposed to default ones clearly speeds up the performance of all transactional memories. The benefit of turning to CF is more important when using trees, which confirms our previous results obtained with our lock-based implementations. In particular, the average speedup for all transactional memories and data structures is of 50%. Interestingly, the speedup of using transaction-based CF does not scale much with the number of threads, probably because the overhead induced by transactional memory and Deuce is too heavy for the contention rise to be visible.

When using transactional memory the benefit of these contention friendly algorithms is apparent just by the fact that abstract operation transactions will have smaller read and write sets causing less contention on the data structure and making the operations less likely to abort. Also structural modifications are each broken into a single transaction causing less contention then they would be if they were include in a single large transaction.

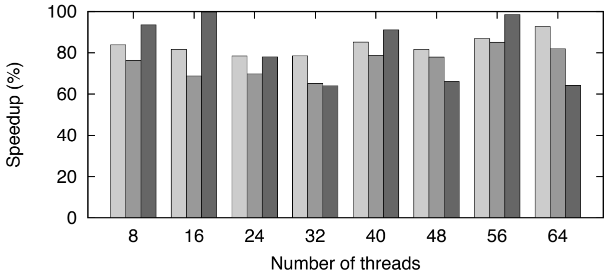
The abstract operations for the tree are very simple, traversals are done in the same way they would be in a sequential algorithm except transactional reads are used. Each physical removal and rotation is performed as a single transaction by the structural adaptation.

In the lock based version of the skip list locks are only used when traversing the bottom level of the structure. Each of the index levels are accessed and modified using only regular read/write operations. This can be applied to the transaction version of the skip list as well. Abstract operation traversals as well as structural modifications to the index level are done outside of transactions. Once an abstract operation traversal has reached the bottom list level a transaction is started, if it arrived on a physically removed node then the operation travels backwards in the list until is reaches a node still in the structure at which point the traversal continues as it would in a sequential list algorithm. Physical removals are done as single transactions by the structural adaptation.

The transactional hash table is very similar to the to the lock-free contention friendly version. Each abstract operation is contained in a single transaction where the compare&swap operations from the lock-free version are replaced by reads and writes. During the rehash operation each bucket rehash is done as a single transaction.



(a) CF-SkipList vs. Pugh Skip List



(b) CF-Tree vs. RB-Tree

Figure 6: STM Speedup when using CF data structures instead of their existing counterparts on the Niagara 2 machine

B Pseudo-code and Description

All three data structures share the general structural adaptation code shown in Algorithm 3.

In the normal case the structural adaptation thread works by performing the *background-struct-adaptation* operation constantly traversing the data structure by calling the *restructuring* procedure. Each iteration of this procedure traverses the entire data structure where at each node it might perform some sort of restructuring or a removal. For some data structures after a complete traversal of the structure is done, some restructuring of the entire structure might be needed, this includes the rehash operation of a hash table or the changing of levels of nodes in the skip list.

For backwards compatibility the structural adaptation can also be distributed among the program threads by calling the *distributed-struct-adaptation* operation. In this case each *insert*/*delete* operation will toss a coin, if the value of this coin is greater then some threshold value the thread will then acquire a global structural adaptation lock, and call the *restructuring* procedure before finally releasing the lock, and continuing with its abstract operation.

B.1 Tree and Skip List

Skip List As with existing skip list algorithms the structure is made up of many levels of linked lists.

The bottom level of is made up of a doubly linked list of nodes. Each node has a *prev* and *next* pointer, as well as pointer to its key k, an integer *level* indicating the number of levels of linked lists this node has, the *rem* and *del* flags, and a lock.

The upper levels are made up of singly linked lists of IndexItems. 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 in the Speculation Friendly 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.

Tree The tree is made up of nodes with each node having left and right child pointers l and r, as well as a pointer to its key k, integers indicating the estimated local height of this node and its children *left-h*, *right-h*, and *local-h*, the *rem* and *del* flags, and a lock.

In addition there is a single pointer *root* that points to the root node of the tree.

B.1.1 Skip List Structural Adaptation

The code for the skip list structural adaptation operations is found in Algorithm 1.

The *restructure-node* procedure takes care of removing marked deleted nodes. For each node it checks if it has both a level of 0 and *del* set to true then tries to remove the node by calling the *remove-node* procedure. This procedure locks both the node to be removed and the node previous to it in the list in order to not conflict with concurrent *insert* and *delete* operations. The node is then simply removed by changing the previous node's pointer to skip the node. Finally the *rem* flag is set to true and the locks are released.

Algorithm 1 Skip List Specific Maintenance Operations

```
restructure-node$(node)_s$:
    if $node.level = 0 \cap node.del$ then
        remove$(node.prev, node)$

restructure-structure$()_s$:
    $size \leftarrow$ raise-node-level()
    $i \leftarrow 1$
    $count \leftarrow$ raise-index-level$(i)$
    while $count > 2$ do
        $i \leftarrow i + 1$
        $count \leftarrow$ raise-index-level$(i)$
    $top \leftarrow first[i]$
    if $\log(size) < i$ then
        lower-index-level()
        // Adjust first array index

lower-index-level$()_s$:
    $index \leftarrow first[2].next$
    while $index \neq \perp$ do
        $index.down \leftarrow \perp$
        $index \leftarrow index.next$

remove$(node, next)_s$:
    if $next.level \neq 0$ then
        return false
    lock$(node)$
    if $node.rem \cup \neg node.del$ then
        unlock$(node)$
        return false
    if $node.next \neq next$ then
        unlock$(node)$
        return false
    lock$(next)$
    $next.next.prev \leftarrow node$
    $node.next \leftarrow next.next$
    $next.rem \leftarrow$ true
    unlock$(node)$
    unlock$(next)$
    return true

raise-index-level$(i)_s$:
    $count \leftarrow 0$
    $prev\text{-}tall \leftarrow first[i+1]$
    $index \leftarrow first[i].next$
    while true do
        $next \leftarrow index.next$
        if $next = \perp$ then
            return $count$
        $prev \leftarrow index.prev$
        if $prev.node.level \leq i$
            $\cap index.node.level \leq i$
                $\cap next.node.level \leq i$ then
            // Allocate a new IndexItem
            // called new
            // Set new as the top IndexItem
            // of index.node
            $new.next \leftarrow prev\text{-}tall.next$
            $prev\text{-}tall.next \leftarrow new$
            $index.node.level \leftarrow i + 1$
            $prev\text{-}tall \leftarrow new$
        $count \leftarrow count + 1$
        $index \leftarrow index.next$
```

The *restructure-structure* procedure raises and lowers the levels of the nodes in order to keep the logarithmic traversal cost of the abstract operations. This is done by calling the *raise-node-level* procedure on the bottom level of the skip list and the *raise-index-level* on higher levels. The code for the procedures is practically the same, just *raise-node-level* is performed on nodes while *raise-index-level* is performed on index levels as such only the *raise-index-level* pseudo code is displayed here. The procedures work by simply traversing the entire level i that they are called on if they encounter 3 or more nodes all with height i then the middle of these nodes is raised to height $i+1$. This is performed on each index level starting from the bottom until there are less than 2 nodes on a level.

Due to nodes being removed from the skip list it might be necessary to decrease the number of index levels in the structure. If the *log* of the number of nodes in the structure is less then the height of the structure then the bottom index level is removed. This is done by the *lower-index-level* procedure which simply traverses the second from bottom index level and sets each index item's *down* pointer to $\perp$. Finally the index of the *first* array must be updated to take account the removal of the bottom index level.

B.1.2 Tree Structural Adaptation

The code for these operations is found in Algorithm 2.

The *restructure-node* procedure takes care of removing marked deleted nodes as well as performing rotations and propagating balance information upwards in the tree.

Like in the skip list only certain nodes are removed. These are the nodes that have 1 or 0 children and are a majority of the nodes in the tree. This avoids expensive removal operations that require finding and moving a successor node.

In order to do a removal first the parent and the node to be removed are locked(in order to prevent conflicts with concurrent *insert* and *delete* operations) and the *del* flag of the node is checked. The node to be removed has its *left* and *right* child pointers changed so that they point to the parent. This is done to ensure a concurrent operation preempted on this node can still proceed. Next the appropriate parent's child pointer is

Algorithm 2 Tree Specific Maintenance Operations

```
restructure-node($node$)$_s$:
  if $node.l.del$ then
    remove($node$, false)
  if $node.r.del$ then
    remove($node$, true)
  propagate($node$)
  if $|node.left\text{-}h - node.right\text{-}h| > 1$ then
    // Perform appropriate rotations

restructure-structure()$_s$:
  // Do nothing

propagate($node$)$_s$:
  if $node.l \neq \bot$ then
    $node.left\text{-}h \leftarrow node.l.localh$
  else
    $node.left\text{-}h \leftarrow 0$
  if $node.r \neq \bot$ then
    $node.right\text{-}h \leftarrow node.r.localh$
  else
    $node.right\text{-}h \leftarrow 0$
  $node.localh \leftarrow$
    max($node.left\text{-}h, node.right\text{-}h$) $+ 1$

remove($parent, left\text{-}child$)$_p$:
  if read($parent.rem$) then
    return false
  if $left\text{-}child$ then
    $n \leftarrow$ read($parent.\ell$)
  else
    $n \leftarrow$ read($parent.r$)
  if $n = \bot$ then
    return false
  lock($parent$)
  lock($n$)
  if $\neg$read($n.deleted$) then
    // release locks
    return false
  if ($child \leftarrow$ read($n.\ell$)) $\neq \bot$ then
    if read($n.r$) $\neq \bot$ then
      // Release locks
      return false
  else
    $child \leftarrow$ read($n.r$)
  if $left\text{-}child$ then
    write($parent.\ell, child$)
  else
    write($parent.r, child$)
  write($n.\ell, parent$)
  write($n.r, parent$)
  write($n.rem$, true)
  // release locks
  update-node-heights()
  return true

right-rotate($parent, left\text{-}child$)$_p$:
  if read($parent.rem$) then
    return false
  if $left\text{-}child$ then
    $n \leftarrow$ read($parent.\ell$)
  else
    $n \leftarrow$ read($parent.r$)
  if $n = \bot$ then
    return false
  $\ell \leftarrow$ read($n.\ell$)
  if $\ell = \bot$ then
    return false
  lock($parent$)
  lock($n$)
  lock($\ell$)
  $\ell r \leftarrow$ read($l.r$)
  $r \leftarrow$ read($n.r$)
  // allocate a node called $new$
  $new.k \leftarrow n.k$
  $new.\ell \leftarrow \ell r$
  $new.r \leftarrow r$
  write($\ell.r, new$)
  write($n.rem$, true)
  if $left\text{-}child$ then
    write($parent.\ell, \ell$)
  else
    write($parent.r, \ell$)
  // release locks
  update-node-heights()
  return true
```

changed to point to the non-null child of the node to be removed (if any). Finally the *rem* flag is set to *true*.

The structural adaptation is also responsible for keeping the tree well balanced. This is done by doing local rotations. Deciding to do a rotation is based on a local estimated height values. The height values are propagated from the leaves to the root by the *propagate* procedure. This procedure is executed per node and simply reads the *l-height* values of its left and right children, before updating its local values and setting its local *l-height* value to 1 greater then the maximum height of its children. If the absolute value of a nodes left and right heights is at least two then an appropriate rotation is performed. Double rotations are performed as two separate single rotations.

In a traditional rotation operation one node is always rotated downwards. If a concurrent traversal operation is preempted on this node then either it might have to abort or rollback in order to ensure it performs a valid traversal or nodes must be locked/marked during traversal.

In order to avoid using locks and aborts/rollbacks, rotations are preformed differently then traditional rotations. A diagram of the new rotation operation is shown in figure **??**. Before performing the rotation the parent node and the node n that will be rotated are locked in order to prevent conflicts with concurrent *insert* and *delete* operations. Instead of actually modifying n, a new node *new* is created that takes n's place in the structure, this node is set have the same values and pointers as n would if a rotation was performed as in existing tree data structures. After the rotation, the node n has its *rem* flag set (to true in the case of a right rotation and by-left-rot in the case of a left rotation) and the locks are released.

The reason for not modifying n is so that concurrent traversals are not set off track. If the node n is removed by a right (resp. left) rotation then its left (resp. right) child has a path to all the nodes as it did before the rotation so a traversal preempted on this node can still traverse the tree safely.

Algorithm 3 States and Restructuring of the Generic CF Algorithm

```
State of process $p$:
    structure, shared pointer to the data
        structure
    frequency, the frequency of a structural
        adaptation

background-struct-adaptation()$_p$:
    while true do
        // continuous background restructuring
        restructuring()

distributed-struct-adaptation()$_p$:
    toss(coin)
    if coin $>$ frequency then
        // restructure now
        restructuring()
    // ...or restructure later

restructuring()$_p$:
    next $\leftarrow$ first-in-trav(structure)
    while next $\neq \perp$ do
        restructure_node(next)
        next $\leftarrow$ next-in-trav(next)
    restructure-structure()
```

B.2 Abstract Operations

The tree and skip list share code for the *contains*, *insert*, *delete* operations displayed in Algorithm 4. These operations might call one of more of the *get_first*, *get_next*, *validate*, *add* procedures which each have specific code for the given data structure, show in Algorithm 6 for the tree and Algorithm 5 for the skip list. None of these additional procedures use locks or other synchronization methods.

Each of the three abstract operations start by calling the *get_first* procedure. This operation returns the root of the tree or the first node of the top index level of the skip list. The operations then traverse the structure using the *get_next* procedure. This procedure either returns the next node in the traversal or $\perp$ if the traversal is done.

The *get_next* procedure traverses the tree by returning the right child if the node's key is lager then k otherwise the left child is returned. If the nodes key is equal to k and the node is not physically removed then $\perp$ is returned. Since locks are not used during traversal the algorithm has to be aware of concurrent rotations. This means returning the right child in case of being preempted on a node that was removed during a left rotation. If the node was removed during a right rotation then the traversal can continue as normal unless it arrives at a child pointer with value $\perp$, in this case it just returns the other child (which is guaranteed to not be $\perp$).

For the skip list the *get_next* procedure traverses the structure just as it would in a sequential algorithm, with the simple exception that is travels backwards in the list using the *prev* pointer in the case of arriving at a node that has been physically removed. If the nodes key is equal to k and the node is not physically removed or if the traversal is at the bottom level and the next node has key greater then k then $\perp$ is returned.

Once $\perp$ is returned *insert* and *delete* operations protect the last node in the traversal by locking it (locking is not necessary for the *contains* operations as it does not make modifications). Due to concurrent operations this node may not longer be the end of the traversal, therefore the *validate* procedure is performed on this node ensuring that the traversal has stopped at the correct location. The validation checks to make sure that the node has not been physically removed and that no new node has been inserted directly after this node.

If the validation succeeds then the traversal is finished. Otherwise the lock protecting the node is released and the traversal continues.

Finally some additional code is executed depending on the operation.

In the case of a *contains* operation, the key and/or the deleted flag of the node is checked and a boolean is returned.

Algorithm 4 Operations of the Generic CF Algorithm

```
State of node $n$:
  $node$ a record with fields:
    $k \in \mathbb{N}$, the node key
    $del \in \{\text{true}, \text{false}\}$, indicate whether
      logically deleted, initially false
    $rem \in \{\text{true}, \text{false}\}$, indicate whether
      physically deleted, initially false
    $lock$, used to lock the node

contains$(k)_p$:
  $node \leftarrow$ get_first$(structure)$
  while true do
    $next \leftarrow$ get_next$(node, k)$
    if $next = \perp$ then
      if validate$(node, k)$ then
        break
    else
      $node \leftarrow next$
  $result \leftarrow$ false
  if $node.k = k$ then
    if $\neg node.del$ then
      $result \leftarrow$ true
  return $result$

insert$(k)_p$:
  $node \leftarrow$ get_first$(structure)$
  while true do
    $next \leftarrow$ get_next$(node, k)$
    if $next = \perp$ then
      lock$(node)$
      if validate$(node, k)$ then
        break
      unlock$(node)$
    else
      $node \leftarrow next$
  $result \leftarrow$ false
  if $node.k = k$ then
    if $node.del$ then
      $node.del \leftarrow$ false
      $result \leftarrow$ true
  else
    add_node$(node, k)$
    $result \leftarrow$ true
  unlock$(node)$
  return $result$

delete$(k)_p$:
  $node \leftarrow$ get_first$(structure)$
  while true do
    $next \leftarrow$ get_next$(node, k)$
    if $next = \perp$ then
      lock$(node)$
      if validate$(node, k)$ then
        break
      unlock$(node)$
    else
      $node \leftarrow next$
  $result \leftarrow$ false
  if $node.k = k$ then
    if $\neg node.del$ then
      $node.del \leftarrow$ true
      $result \leftarrow$ true
  unlock$(node)$
  return $result$
```

Algorithm 5 Skip List Specific Operations

```
Additional fields of IndexItem $item$:
  $IndexItem$ a record with additional fields:
    $next$, pointer to the next IndexItem
      in the SkipList
    $down$, pointer to the IndexItem one
      level below in the SkipList
    $node$, pointer a node in the list at
      the bottom of the SkipList

Additional fields of node $n$:
  $node$ a record with additional fields:
    $next$, pointer to the next node in the list
    $prev$, pointer to the previous node
      in the list
    $level$, integer indicating the level of
      the node, initialized to 0

State of structure $s$:
  $top$, pointer to the first and highest
    level IndexItem in the SkipList
  $first$, array of pointers to the first item
    of each level in the SkipList
  $bottom\text{-}index$ integer indicating the
    level of the bottom IndexItem

get-first$()_s$:
  return $top$

get-next$(node, k)_s$:
  if $node$ is a list node then
    return get-next-node$(node, k)$
  else
    return get-next-index$(node, k)$

get-next-index$(node, k)_s$:
  $next \leftarrow node.next$
  if $next.k > k$ then
    if $node.down \neq \perp$ then
      return $node.down$
    return $node.node$
  else if $next.k = k$ then
    return $next.node$
  return $next$

get-next-node$(node, k)_s$:
  if $node.rem$ then
    while $node.rem$ do
      $node \leftarrow node.prev$
  else
    $next \leftarrow node.next$
    if $next = \perp \cup next.k > k$ then
      return $\perp$
    else
      return $next$

validate$(node, k)_s$:
  if $node.rem$ then
    return false
  if $node.next = \perp \cup node.next.key > k$ then
    return true
  return false

add$(node, k)_s$:
  // allocate a node called new
  $new.key \leftarrow k$
  $new.prev \leftarrow node$
  $new.next \leftarrow node.next$
  $node.next.prev \leftarrow new$
  $node.next \leftarrow new$
```

In the case of the *insert* operation, first the key of the node is checked, if it is equal to the key being search for then the deleted flag of the node is checked (and possibly modified) and a boolean is returned. Otherwise if the key is not equal to the one being searched for then the *add* operation is performed. The code for the *add* operation simply allocates a new node and attaches it to the data structure by modifying a pointer.

In the case of the *delete* operation the key of the node is checked, if it is equal to the key being search for then the deleted flag of the node is checked (and possibly modified) and a boolean is returned. Otherwise false is returned. It should be noted that when these structures are used as maps the deleted flag can be replaced with a pointer

Algorithm 6 Tree Specific Operations

```
Additional fields of node $n$:
  $node$ a record additional with fields:
    $left\text{-}h, right\text{-}h \in \mathbb{N}$, local height of
      left/right child, initially 0
    $\ell, r \in \mathbb{N}$, left/right child pointers,
      initially $\perp$
    $local\text{-}h \in \mathbb{N}$, expected local height,
      initially 1

State of structure $s$:
  $root$, pointer to root

get-first$()_s$:
  return $root$

get-next$(node, k)_s$:
  $rem \leftarrow node.rem$
  if $node.k > k \cup rem = by\text{-}left\text{-}rot$ then
    $next \leftarrow node.right$
  else
    $next \leftarrow node.left$
  if $next = \perp \cap \neg rem$ then
    if $node.k > k$ then
      return $node.left$
    else
      return $node.right$
  return $next$

validate$(node, k)_s$:
  if $node.rem$ then
    return false
  if $node.next.key > k$ then
    $next \leftarrow node.right$
  else
    $next \leftarrow node.left$
  if $next = \perp$ then
    return true
  return false

add$(node, k)_s$:
  // allocate a node called new
  $new.key \leftarrow k$
  if $node.k > k$ then
    $node.right \leftarrow new$
  else
    $node.left \leftarrow new$
```

to the value object (from the key/map pair). When this pointer is set to $\perp$ the node is considered as deleted.

B.3 Hash Table

The hash table is made up of two pointers to tables: *table* and *new_table*. The second is used during resize operations. Each process also keeps local variable *l_pointer* that points to a table.

Each table contains an array, with each location in the array containing a list of nodes. Each location in the array is initialized to point to $\perp$. The array also has a single special node associated with it called the *dummy* node which is used during resizing.

B.3.1 Abstract Operations

The code for these operations is found in Algorithm 7. Each abstract operation starts by calculating the hash value of the key and then calling the *get-first* procedure. This procedure returns the first node in the table located at the bucket given by the hash value or $\perp$ in the case that this bucket is empty. The *get-first* procedure might encounter a dummy node, this means that there is a rehash operation going on that has rehashed this bucket, but not yet finished rehashing the entire table. In this case the local table pointer is updated and the bucket is read again.

Once a value is received from the *get-first* procedure the *contains* simply traverses the list looking for a node with key k.

The *insert* operation also traverses the list looking for a node with key k, if none is found then a new node is allocated and is added to the beginning of the list by performing a compare&swap on the bucket. If the compare&swap fails then the operation restarts.

If the *delete* operation locates a node n with key k in the list then it creates a copy of the list from the first node in the list up to node n but not including n. The bucket is then set to first node of this list by performing a compare&swap. If the compare&swap fails then the operation restarts.

Algorithm 7 HashTable Specific Operations

```
Fields of node $n$:
  node a record with fields:
    $k \in \mathbb{N}$, the node key
    hash, hash value for this node
    next, pointer to next node in list

State of structure $s$:
  table, pointer to array, each location in
    the array contains a list
  table.dummy, pointer to dummy node
  table.mask, binary mask
  new-table, pointer to a table used during
    rehash operations
Process local variables:
  l-table local pointer to table

check-table():
  $t2 \leftarrow new\text{-}table$
  $t1 \leftarrow table$
  if $l\text{-}table = t1$ then
    $l\text{-}table \leftarrow t2$
  else
    $l\text{-}table \leftarrow t1$

get-first$(hash)_s$:
  $l\text{-}table \leftarrow table$
  $node \leftarrow l\text{-}table[hash \& l\text{-}table.mask]$
  while $node = table.dummy$ do
    check-table()
    $node \leftarrow l\text{-}table[hash \& l\text{-}table.mask]$
  return $node$

contains$(k)_s$:
  $node \leftarrow$ get_first(hash$(k)$)
  while $node \neq \bot$ do
    if $k = node.k$ then
      return true
    $node \leftarrow node.next$
  return false

insert$(k)_s$:
  $hash \leftarrow$ hash$(k)$
  while true do
    $first \leftarrow$ get_first$(hash)$
    $node \leftarrow first$
    $index \leftarrow l\text{-}table[hash \& l\text{-}table.mask]$
    while $node \neq \bot$ do
      if $k = node.k$ then
        return false
      $node \leftarrow node.next$
    // allocate a node called new
    $new.k \leftarrow k$
    $new.next \leftarrow first$
    if C&S$(l\text{-}table[index], first, new)$ then
      return $true$

delete$(k)_s$:
  $hash \leftarrow$ hash$(k)$
  while true do
    $first \leftarrow$ get_first$(hash)$
    $node \leftarrow first$
    $index \leftarrow l\text{-}table[hash \& l\text{-}table.mask]$
    while $node \neq \bot$ do
      if $k = node.k$ then
        $prev \leftarrow first$
        $new\text{-}first \leftarrow node.next$
        while $prev \neq node$ do
          // make a copy prev
          if $prev = first$ then
            // set new-first to the copy
          $prev \leftarrow prev.next$
        if $prev \neq first$ then
          $prev.next \leftarrow node.next$
        if C&S$(l\text{-}table[index], first, new\text{-}first)$
          then
          return $true$
        else
          // Goto start of outter while loop
      $node \leftarrow node.next$
    return $true$
```

Algorithm 8 HashTable Specific Maintenance Operations

```
restructure-node$(node)_s$:
  // Do nothing

restructure-structure$()_s$:
  if size$() > threshold$ then
    grow()

size$()_s$:
  $count \leftarrow 0$
  for $i \leftarrow 0$; $i < table.length$; $i{+}{+}$ do
    $next \leftarrow table[i]$
    while $next \neq \bot$ do
      $count \leftarrow count + 1$
      $next \leftarrow next.next$
  return $count$

grow$()_s$:
  $new\text{-}table \leftarrow$ allocate a new table
  for $i \leftarrow 0$; $i < table.length$; $i{+}{+}$ do
    grow-level$(i)$
  $table \leftarrow new\text{-}table$

grow-level$(i)_s$:
  while true do
    $list1 \leftarrow \bot$
    $list2 \leftarrow \bot$
    $next \leftarrow table[i]$
    $first \leftarrow next$
    while $next \neq \bot$ do
      // make a copy of next
      if hash$(next) \& new\text{-}table.mask = i$
        then
        // add copy to list1
      else
        // add copy to list2
      $prev \leftarrow prev.next$
    $new\text{-}table[i] \leftarrow list1$
    $new\text{-}table[i + table.length] \leftarrow list2$
    if C&S$[table[i], first, table.dummy]$ then
      return
```

B.3.2 Structural Adaptations

The code for these operations is found in Algorithm 8. Local node restructuring is not necessary for the hash table.

The structure restructuring consists of two procedures. The first is the *size* operation that simply traverses the structure counting the number of nodes. If the number of nodes has surpassed some threshold then a resize is necessary and the *grow* procedure is called. This procedure starts be creating a new table larger then the previous one by a power of 2. It then goes through the old table rehashing one bucket at a time. At each bucket in the old table it performs the *grow-level* procedure. This procedure makes a copy of each node in the bucket, rehashing them and placing them in their appropriate buckets in the new hash table. The operation then replaces the list in the old table with its dummy node by performing a compare&swap. If the compare and swap fails then the operation is restarted for this level. Once all levels have been rehashed the *table* pointer is modified so that it points to the new table.

C Correctness

Here we present a sketch of the proof that each data structure algorithm satisfies linearizability.

C.1 Skip List

Each of the *contains*, *insert*, and *delete* operations call the *validate* procedure. The *validate* procedure may be called multiple times, but it must return true exactly once. This is used as the linearizability point for the proof sketch. Assume *k* the key provided as input to the abstract operation.

The result of the *contains*, *insert*, and *delete* operations depends on the existence of a node with key *k* in the set described by the data structure. At any single point in time there is exactly one valid location in the list where a node with key *k* can exist (Note that a full proof would require to show this, for example by induction). This location is the *next* pointer of the node in the list with the largest key smaller then *k*. For the purpose of this proof sketch we will use a node (call this node *corr*) that is either the node in the list with key *k* or if no node with key *k* exists the node whose *next* pointer would point to the node with key *k*.

Any operation that modifies a node must lock the node before it performs any modification. Given that the *validate* operation is called while a node is locked it only needs to be shown that the when *validate* returns true the node locked is *corr*.

The nodes in the list are sorted by their keys and the *prev* pointer is not modified when a node is removed so any removed node will always have a path to a non-removed node with a smaller or equal key. This means that there will always be a path from a node with key smaller then or equal to *k* to *corr*. Now since the *get-next* procedure will never traverse past a node that has key larger then *k* the operation it will always have a path to *corr*. If a node that has been removed is reached during traversal the *prev* pointer is followed, otherwise the *next* pointer is followed during traversal so *corr* will be reached eventually.

Before the *validate* operation returns true it first ensures that the locked node has either key *k* or the node pointed to by the locked node's *next* pointer has a key larger then *k*. Second it ensures that the node is not removed ($rem =$ false). Therefore when *validate* returns true, the locked node must be *corr*.

C.2 Tree

The tree is a bit more complicated because traversals have to deal with rotations as well as removals. Like in the skip list the abstract operations can call the *validate* procedure multiple times, but it must return true exactly once. This is used as the linearizability point for the proof sketch. Assume *k* is the key provided as input to the abstract operation.

At any point in time there is exactly one valid location in the tree where a node with key k can exist. This location is the left or right child pointer of a certain node. This pointer points to the node with key k or to $\bot$ if no node with key k exists. For the purpose of this proof sketch we will use a node (call this node *corr*) that is either the node with this pointer (if no node with key k exists) or the node with key k (is a node with key k does exist).

Before the *validate* operation returns true it first ensures that the locked node either has key k or (if the locked node does not have key k) the child pointer from the locked node where k would exist is $\bot$. Second it ensures that the locked node is not removed ($removed = \mathsf{false}$).

Now to complete the sketch it is enough to show that the traversal never passes *corr*. Without rotations or removals this is simple. With removals and rotations the idea is to show that a traversal that is preempted on a node modified by a removal or rotation operation has a path to a node at a higher level in the tree so that the traversal still has a path to *corr*. For removals this is clear due to the fact a removed node has both its child pointers point to its parent during removal. Rotations require a bit more detail. In a traditional left/right rotation one node is rotated downward in the tree while either its left or right child is rotated upwards. For rotations in the contention friendly algorithm the child pointers of the node that would normally be rotated downwards (call this node n) are not modified at all. Instead n is removed from the tree (as the previous parent of n now points to one of n's children) and a new node is created taking n's place. This new node is set up exactly how n would be after a traditional rotation. Now since one of n's children was rotated upwards any concurrent traversal preempted on n will still have a path to all the nodes it did before the rotation.

C.3 HashTable

The linearization of the hash table relies on two things, first that the *next* pointer of a node is never modified after it is set during creation, and second any successful modification to a bucket happens by performing a single *compare&swap* on the bucket's pointer.

The linearization point of the *contains* operation is when the *get-first* procedure reads the first element of the bucket that is later returned. Since the next pointer of nodes is never changed then when the operation traverses the list it observes a valid state of the list. The same is true for *insert* and *delete* operations that complete without performing a *compare&swap*.

If a *compare&swap* is required, then the lineraization point is when the *compare&swap* returns successfully. Given that the *compare&swap* operations are only performed at the first element of the bucket, if the operation succeeds then the lineraiztion is valid because the list at the bucket has not changed since *get-first* procedure read it.

D Garbage Collection

Nodes that are physically removed from the data structures must be garbage collected. In the tree and skip list nodes are physically removed only by the structural adaptations. This can happen during rotations of the tree, during the lowering of levels in the skip list, or during the *remove* operation of either structure. Nodes of the hash table are physically removed by the abstract *delete* operation or by the rehash structural adaptation operation. Once a node is physically removed it will no longer be pointed to by the data structure meaning that no future operation will traverse these nodes.

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. Normally this should be done during the structural adaptation.

E Future Work

E.1 Lock-freedom

The tree and skip list algorithms presented in this paper use locks. By using locks they are susceptible to problems such as a thread crashing or being descheduled while holding a lock. In order to avoid these problems, certain concurrent algorithms such as have been designed to be lock-free such as [11]. Lock-free algorithms are generally considered to be complex and difficult to program. This paper focuses on the methodology of designing contention friendly data structures rather then deep descriptions of the algorithms. For this reason the algorithms are described using locks and a brief intuition on how to make the algorithms lock free is given here.

Lock free algorithms often rely on atomic synchronization primitives such as compare&swap in order to preform tasks without using locks. Often a more complex task will require more then just a single atomic operation. In this case one thread might be required to help another thread's operation so that it completes without blocking other operations.

The *insert*, *delete*, *contains* operations of the contention friendly data structures are simple enough to only require at most one compare&swap operation to complete. For an *insert* this might be performing a compare&swap on a pointer and for a *delete* this might be performing a compare&swap on a flag.

The structural adaptation thread takes care of the more complex operations such as removals and modifications to the structure, operations that might require more then just a single compare&swap. Consider a *remove* operation, the structural adaptation thread will initiate the removal by performing a compare&swap to flag the node letting other processes know that it will be removed. The actual removal is then preformed which requires several more compare&swaps. Before the removal is completed another thread might concurrently traverse the flagged node while searching for some key at a different location in the structure. Like in the lock based algorithms this is fine and the traversal can continue as normal. On the other hand a concurrent traversal might need to perform its operation at the location of the node being removed, for example it might need to insert a new node just after the node. In this case the traversal will help the structural adaptation thread with the removal before completing its operation.

In the lock-free version of the concurrent friendly skip list the structural adaptation thread is in charge of removals and raising and lowers of heights of nodes. The raising and lowering of heights can be done just as in the lock based version since no synchronization is required. The removal of nodes uses the same process as in [11] except the structural adaptation thread will start the removal and program threads will help as necessary.

In the tree removals as well as rotations are performed by the structural adaptation thread and might be helped by program threads. Each of these operations is started by the structural adaptation thread performing a compare&swap to flag the node. The only non-blocking implementation of a binary search tree that we know of is presented in [8]. This tree is *leaf-oriented* where keys are stored in leaf nodes.

Even though we do belive it would be possible to implemnt these lock-free algorithms we do expect them to be very complex and would require more investigation.

E.2 Structural Adaptation Throttling

In the default version of these algorithms a separate structural adaptation thread is created that continually traverses the tree performing structural modification as necessary. In workloads with low update rates this constant traversal will not have any modification to perform, wasting computation. Even if there are extra unused cores this extra computation may be unwanted due to additional power consumption. Future work should include studying way to throttle the structural adaptation dynamically based on the workload. This could mean putting the thread to sleep during periods of low update rate or even starting and stopping the structural adaptation thread entirely. In largely parallel workloads with high update rates it might even be beneficial to have multiple structural adaptation threads that can be started and stopped at will.

E.3 Distributed Structural Adaptation

Each structural adaption is a short local operation, yet each round of structural adaptation is done by a complete traversal of the data structure. Some might argue that if all the hardware of a system is already in use by program threads then why not break the structural adaptation into smaller structural adaptations and distribute them over the abstract modifications. The reason for not doing this is twofold.

Firstly even though each structural adaption is a very local operation, they use global information. For example rotations in a tree need balance information that is propagated from the leaves. Since only nodes with height 1 are removed from the skip list the structural adaptation needs to know about the heights of the other nodes before raising the level of a node in order to ensure that the structure does not become unbalanced. Before resizing the hash table the structural adaptation should know approximately how many nodes are in the table.

Secondly the structural adaptations are in some cases more costly (in terms of computation, not contention) then in existing data structures. For example a rotation requires allocating a new node, choosing the levels of nodes in the skip list requires previously traversing the other nodes to know their height, and resizing the hash table first requires counting the nodes.

Such algorithms with distributed structural adaptation might be possible, but have not been examined here, but could be interesting to study in the future.

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