A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention [⋆]

Jonatan Lindén and Bengt Jonsson

Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{jonatan.linden,bengt}@it.uu.se

Abstract. Priority queues are fundamental to many multiprocessor applications. Several priority queue algorithms based on skiplists have been proposed, as skiplists allow concurrent accesses to different parts of the data structure in a simple way. However, for priority queues on multiprocessors, an inherent bottleneck is the operation that deletes the minimal element. We present a linearizable, lock-free, concurrent priority queue algorithm, based on skiplists, which minimizes the contention for shared memory that is caused by the DeleteMin operation. The main idea is to minimize the number of global updates to shared memory that are performed in one DeleteMin. In comparison with other skiplist-based priority queue algorithms, our algorithm achieves a 30 – 80% improvement.

Keywords: Concurrent Data Structures, Priority Queue, Lock-free, Non-blocking, Skiplist

1 Introduction

Priority queues are of fundamental importance in many multiprocessor applications, ranging from operating system schedulers, over discrete event simulators, to numerical algorithms. A priority queue is an abstract data type, containing a set of key-value pairs. The keys are ordered, and typically interpreted as priorities. It supports two operations: Insert of a given key-value pair, and DeleteMin, which removes the pair with the smallest key and returns its value. Traditionally, priority queues have been implemented on top of heap or tree data structures, e.g., [10]. However, for priority queues that are accessed by large numbers of concurrent processor cores, skiplists [16] are an increasingly popular basis. A major reason is that skiplists allow concurrent accesses to different

[⋆] This work was supported in part by the Swedish Foundation for Strategic Research through the CoDeR-MP project, and by the Swedish Research Council within the UPMARC Linnaeus centre of Excellence.

This is an extended version of the paper with the same name, which was presented at OPODIS 2013: 17th International Conference On Principles Of DIstributed Systems, December 16-18, Nice, France [11].

parts of the data structure in a simple way. Several lock-free concurrent skiplist implementations have been proposed [3, 4, 17].

The performance of skiplist-based data structures can scale well when concurrent threads access different parts of the data structure. However, priority queues offer the particular challenge that all concurrent DeleteMin operations try to remove the same element (viz. the element with the smallest key). This makes DeleteMin the obvious bottleneck for scaling to large numbers of cores. In existing skiplist-based concurrent priority queues [8, 12, 18], the deletion of an element proceeds in two phases: first, the node is logically deleted by setting a delete flag in the node; second, the node is physically deleted by moving pointers in adjacent node(s). Both the logical and the physical deletion involve at least one global update operation, which must either be protected by a lock, or use atomic primitives such as Compare-and-Swap (CAS). Each CAS is expensive in itself, but it also incurs other costs, viz. (i) concurrent CAS operations to the same memory cell cause overhead due to contention, since they must be serialized and all but one will fail, and (ii) any other write or read to the same memory location (more precisely, the same cache line) by another core must be serialized by the coherence protocol, thus generating overhead for inter-core communication. In our experimentation, we have found that the global update operations in the DeleteMin operation are the bottleneck that limits scalability of priority queues. To increase scalability, one should therefore devise an implementation that minimizes the number of such updates.

In this paper, we present a new linearizable, lock-free, concurrent priority queue algorithm, which is based on skiplists. The main advantage of our algorithm is that almost all DeleteMin operations are performed using only a single global update to shared memory. Our algorithm achieves this by not performing any physical deletion of nodes in connection with logical deletion. Instead, our algorithm performs physical deletion in batches when the number of logically deleted nodes exceeds a given threshold. Each batch deletion is performed by simply moving a few pointers in the sentinel head node of the list, so that they point past logically deleted nodes, thus making them unreachable. Thus only one CAS per DeleteMin operation is required (for logical deletion). To enable this batch deletion, we have developed a novel technique to maintain that logically deleted nodes always form a prefix of the skiplist.

Since logically deleted nodes are not immediately physically deleted, this implies that subsequent operations may have to perform a larger number of read operations while traversing the list. However, these reads will be cheap, so that the algorithm overall performs significantly better than previous algorithms, where these reads would conflict with concurrent physical deletions, i.e., writes.

The absence of physical deletion makes our algorithm rather simple in comparison to other lock-free concurrent skiplist algorithms. It is furthermore linearizable: in the paper, we present a high-level proof, and we report on a verification effort using the SPIN model checker, which we have used to verify linearizability by extensive state-space exploration [9, 20].

We have compared the performance of our algorithm to two skiplist-based priority queue algorithms, each of which employs one of the currently existing DeleteMin techniques: (i) a lock-free adaptation of Lotan and Shavit's non-linearizable priority queue [12], which is similar to the algorithm by Herlihy and Shavit [8], and (ii) an algorithm which uses the same procedure for DeleteMin as the algorithm by Sundell and Tsigas [18]. Our algorithm achieves a performance improvement of 30 − 80 % in relation to the compared algorithms, on a limited set of benchmarks. The implementation of the algorithm, and the SPIN model, are both available at `user.it.uu.se/~jonli208/priorityqueue`.

Furthermore, by comparing our algorithm to a specifically designed micro-benchmark, we show that for many cores, it is entirely limited by the logical deletion mechanism in DeleteMin.

In summary, this paper shows a technique for removing scalability bottlenecks in concurrent data structures that are caused by concurrent memory accesses to the same memory location. We show that it is crucial to minimize the number of concurrent global updates, and present a novel internal representation of a skiplist that allows to use only a single global update per DeleteMin operation. We hope that this work will inspire analogous work to develop other concurrent data structures, e.g., concurrent heaps, that minimize conflicts due to concurrent updates.

The paper is organized as follows. In Section 2, we give an overview of related work. In Section 3, we present the main new ideas of our algorithm. In Section 4, we present our algorithm in detail, and prove its correctness in Section 5. The performance evaluation is shown in Section 6. Section 7 contains conclusions. The implementation of the algorithm and the SPIN model are available at `user.it.uu.se/~jonli208/priorityqueue`.

2 Related Work

Skiplists were first proposed for concurrent data structures by Pugh [15, 16], one reason being that they easily allow concurrent modification of different parts of the list. Using skiplists for priority queues was first proposed by Lotan and Shavit [12]. They separated logical and physical deletion, as described in the previous section, which allowed physical deletion of different nodes to proceed in parallel. This still incurs costs of type (ii) (i.e., serializing CAS operations with other accesses to the same memory cell), since many other threads are simultaneously accessing the list. By adding a timestamping mechanism, their algorithm was made linearizable. A lock-free adaptation of this algorithm was presented by Herlihy and Shavit [8], who also observed the contention caused by concurrent physical deletions.

Sundell and Tsigas [18] were first to present a lock-free implementation of a skiplist-based priority queue. Their DeleteMin operation performs logical deletion followed by physical deletion. To achieve linearizability, only a single logically deleted node at the lowest level of the skiplist is allowed at any point in time. Any subsequent thread observing a logically deleted node will help complete the physical deletion. This may reduce latency in some cases, but suffers in general a high cost of both type (i) and (ii). Their skiplist further has back pointers to speed up deletion of elements, and employs back-off to reduce contention of the helping scheme.

Crain et al. [1] propose a technique to reduce memory contention by deferring the physical insertion and removal of logically deleted nodes to a point in the execution when contention is low. In the context of skiplist-based priority queues, this would reduce contention between conflicting global updates, but not reduce the *number* of needed updates, as is done in our algorithm. Thus, they do not reduce the contention between updates and concurrent reads.

Hendler et al. [6] present a methodology, called flat-combining, which reduces contention in arbitrary concurrent data structures (exemplified using a priority queue) that employ coarse-grained locking. In their approach, contended operations are combined into bigger operations, that are handled by the thread currently owning unique access to the structure. Their methodology is orthogonal to ours.

3 Overview of Main Ideas

In this section, we informally motivate and describe our algorithmic invention.

Today's multicore processors are typically equipped with a non-uniform memory architecture, and a cache coherence system, which provides a coherent view of memory to the processor cores. Whenever a core updates a shared memory location, the cache system must first invalidate copies in caches at cores that have previously accessed this location, and afterwards propagate the update to caches in cores that subsequently access the location. The effect is that updates (i.e., writes) cause high latencies if they update memory locations that are accessed by other cores. Thus, a limiting factor for scalability of concurrent data structure is the number of global updates that must be performed to locations that are concurrently accessed by many cores.

Let us now discuss the overhead caused by DeleteMin operations, and how our algorithm reduces it. Skiplists are search structures consisting of hierarchically ordered linked lists, with a probabilistic guarantee of being balanced. The lowest-level list is an ordered list of all stored elements. Higher-level lists serve as shortcuts into lower-level lists, achieving logarithmic search time. Fig. 1a shows a skiplist with 3 elements, having keys 1, 2, and 5. There are sentinel head and tail nodes at the beginning and end. In existing lock-free skiplists [4, 8, 18], a node is deleted by first logically deleting it, by setting a delete flag in it. Thereafter it is physically deleted, by moving pointers in adjacent node(s). In Fig. 1a, nodes 1 and 2 have been logically deleted by setting their delete flags, indicated by black dots. In order to physically remove them, all pointers in the head node and node 1 must be moved. Both logical and physical deletion of a node thus require global update operations, typically CAS-es. During this deletion, many other threads will read the pointers in the first nodes of the list, thus incurring large costs by invalidations and update propagation in the cache coherence system.

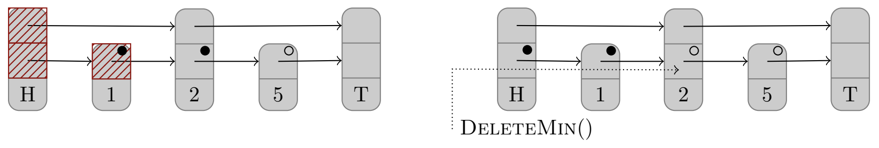

Fig. 1. To the left, memory positions affected by physical deletion of node 1 and 2 in skiplist. To the right, ongoing deletion of node 5 in the new algorithm.

In our new algorithm, we implement the DeleteMin operation *without* performing physical deletion of nodes, in the sense that nodes are never unlinked from the list. Physical deletion is performed in batches, i.e., instead of performing a physical deletion after each logical deletion, we update the pointers in the head node to remove a large number of nodes from the list at a time. The reduction in physical deletion implies that concurrent DeleteMin operations may have to perform a larger number of read operations when traversing the list to find the node to be deleted. However, due to the microarchitecture of today's processors, the cost of these reads, relative to the latencies incurred by an increased number of global writes (e.g., CAS), will be very cheap. A read of a non-modified memory position can be up to 30 times faster than a read of a modified memory position in a multi-socket system [14].

A prerequisite for our scheme is that the logically deleted nodes always form a prefix of the list. This is not the case in existing skiplist-based priority queue algorithms. We have therefore added two mechanisms to our algorithm, whose combined effect is the essence of our algorithmic invention.

1. The delete flag, which signals logical deletion of a node is colocated with the pointer of the *preceding* node (for example in the least-order bit, as detailed by Harris [5]), and not in the deleted node itself.
2. The list contains always at least one logically deleted node (except initially, when no node has yet been deleted).

The first mechanism prevents insertions in front of logically deleted nodes (in contrast to placing the delete flag in the node itself, which prevents insertions after logically deleted nodes), and the second mechanism guarantees that insertions close to the first non-deleted node in the list are safe. Without the second mechanism, Insert would no longer be guaranteed to be correct, as a consequence of the first mechanism.

Fig. 1b shows how the list in Fig. 1a is represented in our algorithm. Note how the delete flags for nodes 1 and 2 are located in the preceding nodes. A typical DeleteMin operation only needs to traverse the lowest level of the skiplist, and set the delete flag of the first node in which it is not already set.

To illustrate how our location of the delete flag prevents insertions in front of logically deleted nodes, Fig. 2 depicts a situation in which nodes 1 and 2 are deleted, and two concurrent threads are active: a DeleteMin operation is

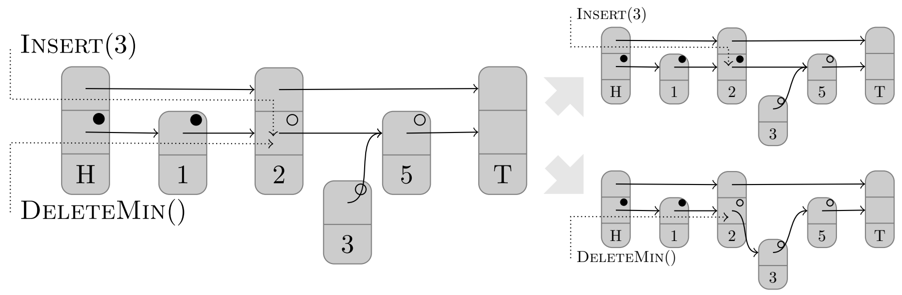

Fig. 2. Concurrent DeleteMin and Insert operation in the new algorithm.

about to set the delete flag in node 2, and an Insert(3) operation is about to move the lowest level pointer of node 2 to point to the new node 3. Two outcomes are possible. If Insert(3) succeeds first, resulting in the situation at bottom right, the DeleteMin operation will just proceed to set the delete bit on the new pointer pointing to the new node 3, which is then logically deleted. If DeleteMin succeeds first, resulting in the situation at top right, the CAS of the Insert(3) will fail, since the pointer in node 2 has changed, thus preventing insertion in front of the logically deleted node 5.

When designing lock-free lists, one must be careful to avoid problems with inconsistent pointer updates when nodes are simultaneously inserted and deleted at the same place in the list. However, in our algorithm we avoid most of these problems, since we do not use fine-grained pointer manipulation to physically delete nodes. A particular consequence is that our algorithm is rather easily seen to be linearizable, in contrast to, e.g., the algorithm by Lotan and Shavit [12], in which timestamps are required to achieve linearizability.

4 The Algorithm

In this section, we describe our algorithm in detail. A skiplist stores a set of key-value pairs, ordered according to the keys. Internally, it consists of a set of hierarchically ordered linked lists. The lowest-level list (i.e., level 0) is a complete ordered list of all stored key-value pairs, which also defines the logical state of the skiplist. Higher level linked lists serve as shortcuts into lower levels. Thus, a skiplist with only one level is simply an ordinary linked list. We assume unique keys, but duplicates can be handled as detailed in previous work [4, 18].

A node in the skiplist is a structure, as described in Listing 1, which contains a value, a key, and an array of next pointers, which contains one pointer for each level in which the node participates. Each node also contains a delete flag `d`, which is true if the successor of the node is deleted from the logical state of the list. In the actual implementation, the delete flag is stored together with the lowest-level next pointer, in its least-order bit, so that atomic operations can be performed on the combination of the lowest-level next pointer and the delete flag. In our pseudocode, we use the notation $\langle x.\text{next}[0], x.\text{d}\rangle$ to denote the combination of a pointer x.next[0] and a delete flag x.d, which can be stored together in one word. Each node also has a flag `inserting`, which is true until INSERT has completed the insertion.

Listing 1 Skiplist and node structures.

```
structure node_t:
    value_t value
    key_t key
    bool d
    bool inserting
    node_t *next[]

structure skiplist_t:
    node_t *head
    node_t *tail
    integer nlevels
```

Algorithm 2 Deletion of minimal element in new algorithm.

```
function DeleteMin(skiplist_t $q$)
    $x \leftarrow q$.head, $offset \leftarrow 0$, $newhead \leftarrow$ Null, $obshead \leftarrow x$.next[0]
    repeat
        $\langle nxt, d\rangle \leftarrow \langle x.\text{next}[0], x.\text{d}\rangle$          // Peek forward.
        if $nxt = q$.tail then                  // If queue is empty, return.
            return Empty
        if $x$.inserting and $newhead$ = Null then
            $newhead \leftarrow x$                    // Head may not surpass pending insert.
        $\langle nxt, d\rangle \leftarrow$FAO(&$\langle x.\text{next}[0], x.\text{d}\rangle$, 1)  //Logical deletion of successor to x.
        $offset \leftarrow offset + 1$
        $x \leftarrow nxt$                       // Traverse list to next node.
    until not $d$                            // If delete bit of x set, traverse.
    $v \leftarrow x$.value                        // Exclusive access to node x, save value.
    if $offset <$ BoundOffset then return $v$
    if $newhead$ = Null then $newhead \leftarrow x$
    if CAS(&$\langle q.\text{head.next}[0], q.\text{head.d}\rangle$, $\langle obshead, 1\rangle$, $\langle newhead, 1\rangle$) then
        Restructure($q$)                     // Update head's upper level pointers.
        $cur \leftarrow obshead$
        while $cur \neq newhead$ do          // Mark segment for memory reclamation.
            $nxt \leftarrow cur$.next[0]
            MarkRecycle($cur$)
            $cur \leftarrow nxt$
    return $v$
```

The DeleteMin operation. The DeleteMin operation is shown in Algorithm 2. In the main **repeat-until** loop, it traverses the lowest-level linked list from its head, by following next pointers, searching for the first node not having its delete flag set. At line 5, it is checked whether the successor of the current node points to the dummy tail node, in which case DeleteMin returns Empty. Otherwise, the delete flag is unconditionally set using an atomic Fetch-And-Or[1] (FAO) instruction (at line 9). If the updated node's delete flag was set before the FAO (inspected at line 12), its successor was already deleted, and the traversal continues to the next node. Otherwise, the node's successor has been successfully deleted, and the successor's value may safely be read (at line 13).

After the traversal, the DeleteMin operation checks whether the prefix of logically deleted nodes (measured by variable *offset*) has now become longer than the threshold BoundOffset. If so, DeleteMin tries to update the `next`[0] pointer of q`.head` to the value of *newhead*, using a CAS instruction; it maintains the invariant that all nodes that precede the node pointed to by *newhead* are already logically deleted. It should be noted that *newhead* will never go past any node with `inserting` set to true (i.e., in the process of being inserted), as ensured by lines 7 and 15. If the CAS is successful, this means that DeleteMin has succeeded to physically remove a prefix of the lowest-level list. If the CAS is unsuccessful, then some other DeleteMin operation has started to perform the physical deletion, and the operation returns. In the successful case, the operation must proceed to update the higher level pointers, which is done in the Restructure operation, shown in Algorithm 3. After the completion of the Restructure operation, the DeleteMin operation proceeds to mark the nodes between the observed first node, *obshead*, and the *newhead* node, as ready for recycling (lines 19 – 22).

Algorithm 3 Restructure operation.

```
1  function Restructure(skiplist_t $q$)
2      $i \leftarrow q$.nlevels−1, $pred \leftarrow q$.head
3      while $i > 0$ do
4          $h \leftarrow q$.head.next[$i$]
5          $cur \leftarrow pred$.next[$i$]
6          if not $h$.d then
7              $i \leftarrow i - 1$
8              continue
9          while $cur$.d do
10             $pred \leftarrow cur$
11             $cur \leftarrow pred$.next[$i$]
12         if CAS (&$q$.head.next[$i$], $h$,
                   $pred$.next[$i$]) then
13             $i \leftarrow i - 1$
```

The Restructure operation updates the pointers of q`.head`, except for level 0, starting from the highest level. A traversal is initiated from the top level head pointed node. At each level, the state of the head's pointer at the current level is first recorded, at line 4, and considered for updating only if the node pointed to has become deleted. Thereafter, that level is traversed until encountering the first node whose successor does not have the delete flag set, stored in variable *pred*, at lines 9 – 11. The head node's next pointer at that level is then updated by means of a CAS, at line 12. If successful, the traversal continues one level down from *pred*. If the CAS operation fails, the same procedure will then be repeated for the same level, also here continuing from *pred*.

[1] Fetch-and-Or is an instruction that atomically updates a memory location using or, and returns the previous value. It can be emulated using CAS, and is used here to simplify presentation.

Algorithm 4 Insertion of node with priority k.

```
1  function INSERT(skiplist_t $q$, key_t $k$, value_t $v$)
2      $height \leftarrow$ RANDOM(1, $q$.nlevels), $new \leftarrow$ ALLOCNODE($height$)
3      $new$.key $\leftarrow k$, $new$.value $\leftarrow v$, $new$.d $\leftarrow 0$, $new$.inserting $\leftarrow 1$
4      repeat
5          ($preds$, $succs$, $del$) $\leftarrow$ LOCATEPREDS($q, k$)
6          $new$.next[0] $\leftarrow succs$[0]                //Prepare new to be inserted.
7      until CAS(&⟨$preds$[0].next[0], $preds$[0].d⟩, ⟨$succs$[0], 0⟩, ⟨$new$, 0⟩)
8      $i \leftarrow 1$
9      while $i < height$ do                          //Insert node at higher levels.
10         $new$.next[$i$] $\leftarrow succs$[$i$]             //Set next pointer of new node.
11         if $new$.d or $succs$[$i$].d or $succs$[$i$] $= del$ then
12             break                                  //new already deleted, finish.
13         if CAS(&$preds$[i].next[i], $succs$[i], $new$) then
14             $i \leftarrow i + 1$                   //If success, ascend to next level.
15         else
16             ($preds$, $succs$, $del$) $\leftarrow$ LOCATEPREDS($q, k$)
17             if $succs$[0] $\neq new$ then break     //New has been deleted.
18     $new$.inserting $\leftarrow 0$                  //Allow batch deletion past this node.
```

The INSERT **operation.** The INSERT operation is similar to a typical concurrent insert operation in skiplist algorithms [4, 8, 18]. The main difference is that the logically deleted prefix and the separation of the delete flag and the node has to be taken into account. Recall that we assume keys to be unique, which implies that the parameter k is guaranteed not to be in the skiplist when INSERT is called. The operation (Algorithm 4) works as follows. The new node is first initialized at lines $2 - 3$. Thereafter, a search for the predecessor nodes, at each level, of the position where the new key value is to be inserted, is performed by the operation LOCATEPREDS (at line 5). The LOCATEPREDS operation itself is shown in Algorithm 5.

Once the candidate predecessors and successors have been located, the new node is linked in at the lowest level of the skiplist, by updating the lowest-level predecessor's next pointer to point to *new* using CAS, at line 7. Note that this pointer also contains the delete flag: this implies that if meanwhile *succs*[0] has been deleted, the delete bit in *preds*[0].`next`[0] has been modified, and the CAS fails. If the CAS is successful, then INSERT proceeds with insertion of the *new* node at higher levels. This is done bottom up, so that a node is visible on all lower levels before it is visible on a higher level. As a consequence, it is always possible to descend from a level $i + 1$ list to a level i list.

The insertion at higher levels proceeds level by level, going from lower to higher levels. For each level i, it first sets the *new* node's `next`[i] pointer to point to the candidate successor, at line 10. Thereafter, it checks whether the lowest-level successor of the *new* node has been deleted since *new* was inserted, at line 11, or if the candidate successor is deleted. In the former case, it is implied that *new* is deleted as well. In the latter case, the result from LOCATEPREDS is possibly *skewed*, as depicted in Fig. 3, and elaborated on in the next paragraph. In both cases the operation is considered completed, and it returns immediately. Otherwise, the level i insertion is attempted at line 13. If successful, the insertion ascends to the next level. If unsuccessful, predecessors and successors will be recorded anew (line 16), and the insertion procedure will be repeated for level i. When INSERT is done, completion of the insertion is signaled to other threads by setting *new*.`inserting` to 0, at line 18.

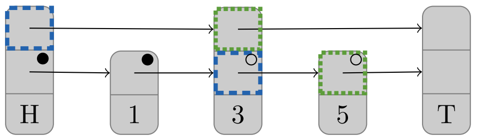

Fig. 3. A skewed search result at the completion of LOCATEPREDS$(q, 2)$. The *preds* (dashed) are overlapping with the *succs* (dotted).

The skewed search results mentioned above occur due to the separation of a node and its delete status. The delete flag is located in the predecessor node, and knowledge about the predecessor is only available in the lowest level. Hence, when LOCATEPREDS searches for predecessors and successors to a key, at higher levels, a node not having the delete flag set must be assumed to be non-deleted, and thus qualifies as a successor. When the lowest level is reached, the node may in fact turn out to be deleted, and the traversal will thus continue past the higher level successor, until the correct position is found at the lowest level. This results in $preds[i]$ and $succs[i]$, for all levels $i > 0$, being skewed in relation to $preds[0]$ and $succs[0]$. The situation is depicted in Fig. 3. The skew may also occur at arbitrary levels, if the deleted prefix is extended during an ongoing LOCATEPREDS operation.

The LOCATEPREDS operation (Algorithm 5), locates the predecessors and successors of a new node that is to be inserted. Starting with the highest-level list, each list is traversed, until a node with a greater key is found, which is also either *a*) a node not having its delete flag set if the level is greater than 0, or *b*) a non-deleted node at the lowest level (lines 6 – 10). When such a node is found, it and its predecessor are recorded, and the search descends to the level below, or is completed. The last

Algorithm 5 LOCATEPREDS operation.

```
1  function LocatePreds(skiplist_t $q$,key_t $k$)
2      $i \leftarrow q$.nlevels$-1$,$pred \leftarrow q$.head
3      $del \leftarrow$NULL
4      while $i \geq 0$ do
5          $\langle cur, d\rangle \leftarrow \langle pred.\text{next}[i], pred.\text{d}\rangle$
6          while $cur$.key $< k$ or $cur$.d or
           ($d$ and $i = 0$) do
7              if $d$ and $i = 0$ then
8                  $del \leftarrow cur$
9              $pred \leftarrow cur$
10             $\langle cur, d\rangle \leftarrow \langle pred.\text{next}[i], pred.\text{d}\rangle$
11         $preds[i] \leftarrow pred$
12         $succs[i] \leftarrow cur$
13         $i \leftarrow i - 1$
14     return ($preds$, $succs$, $del$)
```

(at the moment of traversal) deleted node of the deleted prefix, is recorded in *del*, if traversed (line 7). It is then returned together with *preds* and *succs*.

Memory management. The memory of deleted nodes has to be safely deallocated. This is particularly difficult for non-blocking algorithms, since a thread may be observing outdated parts of the shared state. We use Keir Fraser's epoch based reclamation (EBR) [4] to handle memory reclamation. In short terms, EBR works as follows. Each thread signals when it enters and leaves an operation that accesses the skiplist. After a node has been marked for memory reclamation by some thread, the node will be deallocated (or reused) only when all threads that may possibly have a (direct or indirect) reference to it have returned. Since in our algorithm, a node is marked for recycling only when it can no longer be reached from any pointer of form q.`head`.`next`[i], we guarantee that any thread that enters after the marking, cannot reach a node that is recycled.

One property of EBR is that, even though it is very fast in general, the memory reclamation is not truly non-blocking. A thread blocking in the middle of an operation, i.e., after it has signaled entry, but before signaling exit, could have (and must therefore be assumed to have) a reference to any node in the queue, thereby effectively blocking reclamation of any nodes deleted after that point in time. Since we focus on global performance, we do not consider this a problem. If it is important to have non-blocking memory reclamation, there are other standard solutions, such as hazard pointers [13] or reference counting [19] for memory reclamation.

5 Correctness and Linearizability

In this section, we establish that our algorithm is correct, i.e., it is a linearizable [7] implementation of a priority queue, it is lock-free, and all memory is safely reclaimed. The linearizability of the algorithm has also been checked using the SPIN model checker. The code for the SPIN model is listed in the appendix (it is also available at the companion website of the paper).

Let us consider the (global) state of the algorithm at some point during its execution. By a *node*, we mean a node of type `node_t` that has been allocated at some previous point in time. A *head-pointed node* is a node that is pointed to by a head pointer (i.e., a pointer of the form q.`head`.`next`[i]). A *live node* is a node that is reachable by a sequence of 0 or more pointers of form `next`[i] from a head-pointed node. A *deleted node* is a node that is pointed to by a lowest-level pointer (of form `next`[0]) from a node, which has its delete flag set to 1. A *recycled* node is a node that has been marked for recycling by MarkRecycle.

Linearizability. We first establish an invariant of the algorithm. Thereafter, we prove that the algorithm is a linearizable implementation of a priority queue. The invariant consists of three parts: part 1 specifies the structure of the lowest-level list, part 2 specifies the postcondition of the LocatePreds operation, and part 3 specifies the structure of the higher level lists.

1. **Structure of lowest-level list**
   (a) The set of live nodes and the `next`[0]-pointers form a linear singly linked list, starting at the first head-pointed node, and terminated by the node q.`tail`. Initially, the list does not contain any deleted nodes. After the first invocation of DELETEMIN, q.`head.next`[0] always has the delete flag set.
   (b) The live deleted nodes form a strict prefix of the live nodes (recall that the tail node is live and never deleted).
   (c) The live non-deleted nodes have increasing keys.
2. LOCATEPREDS **postcondition**
   When LOCATEPREDS(q, k) returns, for any i, the following conditions hold:
   (a) $preds[i]$.`key` $< k$, or $preds[i]$ is deleted.
   (b) $succs[i]$.`key` $\geq k$.
   (c) $preds[i-1]$ is reachable from $preds[i]$ via a (possibly empty) sequence of `next`$[i-1]$ pointers, or $preds[i]$ is the head node.
   (d) If for some $j > 0$, $succs[0]$ is reachable from $succs[j]$ by a non-empty sequence of `next`[0] pointers, then either $succs[j]$.`d` $= 1$ or $del = succs[j]$.
3. **Structure of higher-level lists**
   For any live node n, and for any $i > 0$, if n is pointed to by a `next`$[j]$ pointer for some $j \geq i$ and n.`next`$[i]$ is non-null, then n.`next`$[i]$ points to a node which is reachable from n by a non-empty sequence of `next`[0] pointers.

Proof. **1a)** Since nodes are never unlinked from the list, the invariant follows by noting that an insertion into the lowest-level list (at line 7 of INSERT) always inserts a new node between two contiguous ones; this follows by noting that the CAS succeeds only if no additional node has been inserted between $preds[0]$ and $succs[0]$. Initially the only nodes are the sentinel head and the tail nodes, such that the tail is the successor of the head. After the first DELETEMIN operation that does not return EMPTY, which sets the delete flag in q.`head.next`[0] (at line 9), the only line that modifies q.`head.next`[0] is line 16 in DELETEMIN, always keeping the delete flag set at each write.

1b) Statements that may affect this invariant are:
   (i) DELETEMIN, line 9, which logically deletes x.`next`[0]: since x is either deleted or the head node, DELETEMIN may only extend an existing prefix of deleted nodes, and
   (ii) INSERT, line 7, which inserts a node: LOCATEPREDS and the semantics of CAS guarantee that $preds[0]$.`next`[0] does not have its delete flag set, i.e., the node following the inserted one is not deleted.
   (iii) DELETEMIN, line 16, which updates the head's lowest-level pointer: the delete bit is always set in the CAS. Thus, the first head-pointed node remains deleted.

1c) By postconditions 2a and 2b, when LOCATEPREDS returns, having been called with key k as argument, we have that (i) $k < succs[0]$.`key`, and (ii) $preds[0]$.`key` $< k$ or $preds[0]$ is deleted. The lowest-level insertion of a new node at line 7 of INSERT is completed only if $preds[0]$.`next`[0] and $succs[0]$ point to the same node, and if $succs[0]$ is non-deleted. If $preds[0]$.`key` $> k$, then *preds*[0] is deleted, and the inserted node becomes the first non-deleted node.

2a) Initially, when the traversal starts in LocatePreds, the first node (q.`head`), which is also the first potential predecessor, has a key smaller than k, by definition. The *preds* and *succs* pointers are recorded when the condition for the **while** loop in LocatePreds at line 6 no longer is true. The condition for *preds*[i] is defined by the next to last iteration before the while loop stops, as given by line 9. Hence, for *preds*, the condition is still true, and we get *preds*[i].`key` $< k$ or *preds*[i].`d` $= 1$, or $i = 0$ (the traversal has reached the bottom level), and the predecessor to *preds*[0] has the delete flag set. In either case, if not *preds*[i].`key` $< k$, then *preds*[i] is deleted.

2b) Follows directly from the negation of the condition for the **while** loop at line 6 in LocatePreds.

2c) That *preds*[$i-1$] is reachable from *preds*[i] via a sequence of `next`[$i-1$] pointers follows by the mechanism for traversing nodes at lines 6 – 10 in LocatePreds. We also observe that the head node may be updated by the Restructure operation, after LocatePreds have returned, rendering *preds*[$i-1$] unreachable from *preds*[i], i.e., the head node.

2d) If *succs*[0] is reachable from *succs*[j], for $j > 0$, then, at some point during the traversal in LocatePreds, at some level $i < j$, *succs*[j] must have been passed (not necessarily traversed), for the reason that some n, which is reachable from *succs*[j] by a (possibly empty) sequence of `next`-pointers, either (i) has the delete flag n.`d` set, or (ii) n.`key` $<$ *succs*[j].`key`. In both cases it means that *succs*[j] is deleted, in the latter case by invariant 1c. If *succs*[j].`d` is not set, but *succs*[j] itself is deleted, then clearly *del* = *succs*[j], as guaranteed by line 7, where the last (at the moment of traversal) deleted node of the deleted prefix is recorded.

3) By invariant 1a, the `next`[0] pointers organize the live nodes into a linear structure. We observe that reachability between live nodes by a sequence of `next`[0] pointers is preserved by all statements of the algorithm. We must therefore check that the invariant is established when higher-level pointers are updated. This happens in two places in Insert, when a new node is inserted at higher levels.

- At line 10 in Insert. When this line is executed, the node *new* is not yet reached by any `next`[j] pointer for $j \geq i$. This happens only if the condition on the next line 11 is true. Hence, by postcondition 2d, *succs*[0] is not reachable from *succs*[i]. As both *succs*[0] and *succs*[i] are part of the lowest-level list, this implies that *succs*[i] is reachable from *succs*[0]. Since *succs*[0] clearly is reachable from *new*, the invariant follows.
- At line 13 in Insert. By repeated application of postcondition 2c, the inserted node *new* is reachable from *preds*[i] via a sequence of `next`[0] pointers when LocatePreds returns, and consequently also when line 13 in Insert is executed.

We can now establish that our algorithm is a linearizable implementation of a priority queue. Recall that a priority queue can be abstractly specified as having a state which is an ordered sequence of (key,value)-pairs, and two operations: DeleteMin, which removes the pair with the smallest key value, and Insert, which inserts a (key,value)-pair at the appropriate position.

To establish that our algorithm is a linearizable implementation, we first define the *abstract state* of our implementation, at any point in time, to be the sequence of (key,value)-pairs in non-deleted live nodes, connected by `next`[0] pointers. By invariant 1c the keys are increasing. To prove linearizability, we must then for each operation specify a *linearization point*, i.e., a precise point in time at which the implemented operation affects the abstract state. These are as follows.

DeleteMin The linearization point is at line 9 of DeleteMin in the case that the operation succeeds: clearly the abstract state is changed in the appropriate way (invariant 1b). In the case of an unsuccessful DeleteMin, it linearizes at line 4, if the statement assigns q.`tail` to *nxt* (this is discovered at line 5).

Insert The operation is linearized when the new node is inserted at line 7 of Insert: clearly the new node is inserted at the appropriate place. Note that Insert cannot fail: it will retry until successful.

Memory reclamation. We now establish an invariant that affects the memory reclamation. We consider the memory reclamation to be correct if all memory is eventually reclaimed safely, i.e., memory is reclaimed when it no longer will be accessed by any thread. The invariant consists of two parts: part 1 defines the structure of non-live nodes, part 2 specifies the postcondition of the Restructure operation.

4. **Structure of non-live nodes**
   (a) An inserted node n such that n.`inserting` is set, is live.
   (b) Only non-live nodes are marked for recycling.
   (c) The non-live nodes are partitioned into sets of nodes, such that for each set, either (i) all nodes in the set have been marked for recycling, or (ii) a thread is allotted to the set, which currently executes lines 19 – 22 of DeleteMin, and is in the process of marking all nodes in the set for recycling.
5. Restructure **postcondition**
   When Restructure(q) returns, then for any $i > 0$, q.`head.next`[i] is reachable from the node pointed to by q.`head.next`[0] at the start of the invocation.

4a) By inspecting the statements at lines 7 – 8 of DeleteMin, together with the CAS at line 16, we see that the q.`head.next`[0] will not point past n.

4b) Follows from observing that when DeleteMin reaches line 19, then by postcondition 5, all nodes preceding *newhead* are non-live. Clearly, *newhead* is reachable from *obshead*, and the **while** loop hence terminates at *newhead*. Furthermore, by invariant 4a, the head will not be modified to point to an inserting but non-live node.

4c) Nodes are made non-live in DeleteMin at lines 16 and 17, as detailed by postcondition 5. The thread for which the CAS at line 16 succeeded, gains exclusive access (among DeleteMin threads performing memory reclamation) to the segment of non-live nodes between the nodes pointed to by local variables *obshead* and *newhead*, including the node *obshead*. The segments are disjoint since *obshead* of a DeleteMin invocation will equal *newhead* of a preceding DeleteMin invocation, as given by lines 2 and 16.

5) Let n be the node to which $q.\texttt{head.next}[0]$ points when the Restructure operation starts. We note that, after the first invocation of DeleteMin, $q.\texttt{head.next}[0]$ always points to a deleted node (If no node has been deleted, then the condition is trivially true). We also note that there is at most one deleted node not having its delete flag set, as a consequence of invariant 1b. Now, for any $i > 0$, either the head-pointed node n' at level i does not have the delete flag set, in which case the Restructure does not update $q.\texttt{head.next}[i]$. Since the delete flag is not set, either n' is deleted, in which case it may be equal to n, or it is not deleted, in which case it is clearly reachable from n, by invariant 1b. In the case $n'.\texttt{d} = 1$, level i will be traversed until a node not having its delete flag is observed, at lines 9 to 11. Using the same reasoning as in the previous case, we reach the conclusion that the observed node is reachable from n.

We can now conclude that memory reclamation is done safely, i.e, only nodes not reachable from any pointer of the form $q.\texttt{head.next}[i]$ are marked for recycling (by invariant 4b), subsequently being recycled by epoch-based reclamation when all threads that have observed a previous state of the head node have returned. Furthermore, all memory is reclaimed, by invariant 4c.

Lock-freedom. We now briefly show that the algorithm is lock-free, i.e., for each operation, infinitely often some invocation finishes in a finite number of steps. All modifications of the data structure are performed using CAS (except for clearing the `inserting` flag), and for each operation to finish, only a bounded number of successful CAS instructions are needed. Whenever a CAS succeeds in an operation, that operation has made progress, and if an operation's CAS fails at some point, then it means that another operation has made progress.

Even though the algorithm is lock-free, there is one situation in which the performance may be deteriorated by a single blocking thread. A thread that blocks during an Insert operation, causes the `inserting` flag of the corresponding node to remain set for an indefinite duration. Thereby it prevents $q.\texttt{head.next}[0]$ from being updated to point beyond that node. This may eventually lead to a very long (but finite) prefix of live deleted nodes, that must be traversed by all DeleteMin operations.

6 Performance Evaluation

In this section, we evaluate the performance of our new algorithm in comparison with two lock-free skiplist-based priority queues, representative of two different

DeleteMin approaches. We also report on the impact of the choice of the parameter BoundOffset. Finally, we relate the algorithms performance to the limits imposed by the necessity of performing one global update per DeleteMin operation.

The compared algorithms are all implemented on top of Keir Fraser's [4] skiplist implementation, an open source state of the art implementation. Since the same base implementation is used for all compared algorithms, the performance differences between the implementations are directly related to the algorithms themselves, and hence, the DeleteMin operations. The two compared algorithms are respectively based on:

– *SaT* – Sundell and Tsigas' linearizable algorithm [18], in which only the first node of the lowest-level list in the skiplist may be logically deleted, at any given moment. If a subsequent thread observes a logically deleted node, it will first help with completing the physical deletion. The algorithm implemented here is a simplified version of their algorithm, neither using back link pointers nor exponential back-off, but it uses the same DeleteMin operation.
– *HaS* – Herlihy and Shavit's non-linearizable lock-free adaptation [8] of Lotan and Shavit's algorithm [12], in which deleted nodes need not be a prefix. Logical deletion is performed using atomically updated delete flags. Physical deletion is initiated directly after the logical deletion. Insertions are allowed to occur between logically deleted nodes.

The algorithms are evaluated on two types of synthetic microbenchmarks, and as a part of one real application, URDME [2]. The benchmarks are:

– *Uniform* – Each thread randomly chooses to perform either an Insert or a DeleteMin operation. Inserted keys are uniformly distributed. This is the de facto standard when evaluating concurrent data structures [5, 12, 18].
– *DES* – The second synthetic benchmark is set up to represent a discrete event simulator (DES) workload, in which the priorities represent the time of future events. Hence the key values are increasing: each deleted key generates new key values that are increased by an exponentially distributed offset.
– *URDME* – URDME [2] is a stochastic DES framework for reaction-diffusion models. In this benchmark, the priority queue is used as the event queue in URDME. A model with a queue length of 1000 has been simulated.

The experiments were performed on two machines, (i) a 4-socket Intel Xeon E5-4650 machine, of which each socket has 8 cores and a 20 MB shared L3-cache, and (ii) a 4-socket AMD Opteron 6276 machine, of which each socket has 8 "modules" (containing two separate integer pipelines, that share some resources), and a 16 MB shared L3-cache. In addition, every two modules share a 2 MB L2-cache. The compiler used was GCC 4.7.2, at O3 optimization level. Each benchmark was run 5 times, for 10 seconds each time, of which the average throughput is presented. The number of threads varied between 1 to 32. The threshold for updating the head node, BoundOffset, was chosen in accordance to the maximum latency of the setup, with a higher value for a larger number

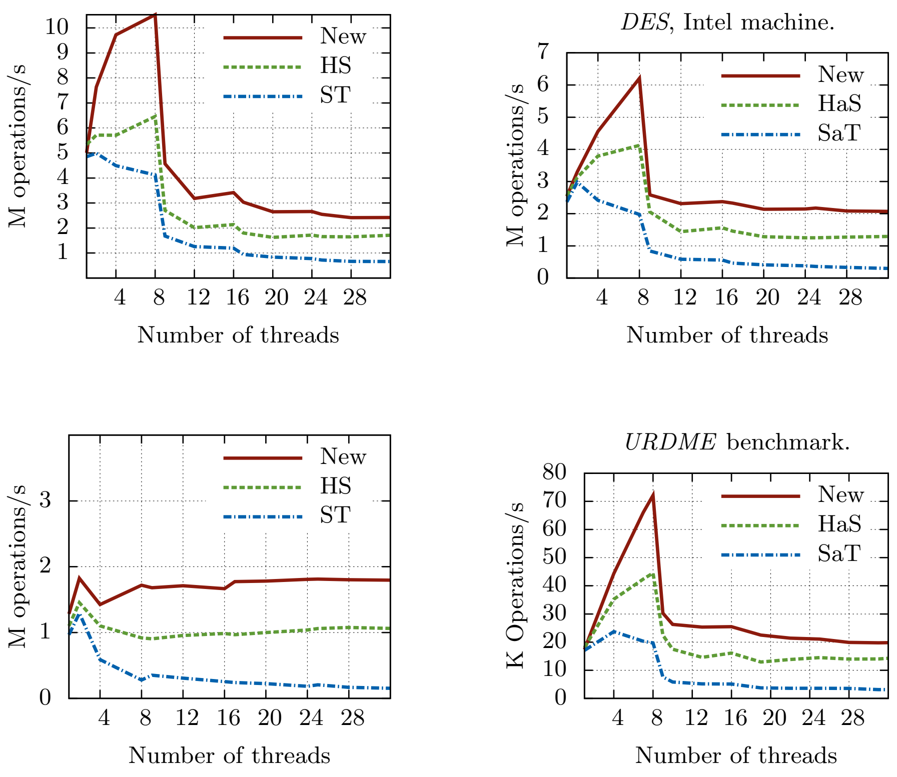

Fig. 4. Throughput of algorithms for the benchmarks.

of threads, ranging from 1 to 64. Threads were pinned to cores, to reduce the variance of the results and to make cache coherence effects visible.

The scaling of the different priority queues is shown in Fig. 4. We see that in general, our new algorithm is between 30 – 80% faster. The largest performance improvement is seen in the *DES* benchmark, on the Intel machine. The performance improvement of the *Uniform* benchmark is slightly smaller. We note that when using uniformly distributed keys, a majority of the inserts occur close to the head, and as a consequence the CAS instructions are more prone to fail, causing retrials of the operations. The penalty of the retry-search loop is slightly higher in our case, because of the traversal of the prefix of logically deleted nodes.

We note a steep drop in performance going from 8 to 9 threads on the Intel machine. This is a direct effect of threads being pinned to cores: up to 8 threads all threads share the L3-cache. At 9 threads or more, communication is done across sockets, with increased latency as a consequence. Likewise, for the AMD machine: every 2 threads share an L2-cache, outside which communication is more expensive.

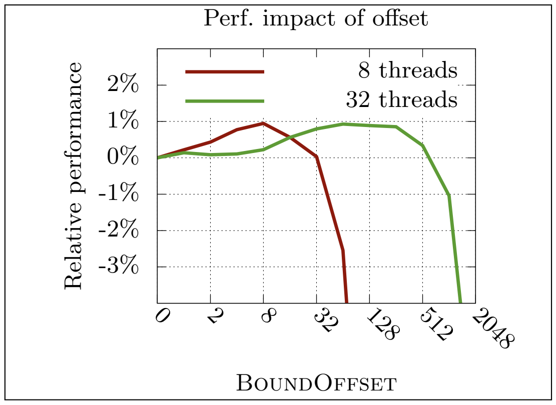

Fig. 5. Effect of BoundOffset on performance. The benchmark was run on the Intel machine.

Impact of BoundOffset. The choice of the parameter BoundOffset affects the performance, weighing CAS contention against the cache and memory latency. When BoundOffset has a low value, indicating that threads will try to update the head node's pointers more often, the probability that threads will contend for updating the lowest-level head pointer increases. When it has a high value, the number of nodes in the prefix that must be traversed for each operation increases, thus becoming more sensitive to the latency of memory.

In Fig. 5 we see that the effect of memory latency far outweighs that of the CAS contention. As the offset increases, performance increases marginally, as a result of decreased CAS contention and cachelines being invalidated less often. At a certain offset, the performance drops drastically, indicating where the total latency of the reads surpasses the latency incurred by updating the head pointers. The main performance increase of the algorithm can hence be concluded to come from the fact that only a single thread performs physical deletion, i.e., there is no contention for updating the head's next pointers. That several nodes are physically deleted at the same time is less important.

Performance limitations. We investigated the scalability bottleneck of the algorithm. For this goal, we devised a microbenchmark in which n threads access a data structure consisting of n nodes in a circular linked list. Each thread k traverses $n - 1$ nodes, and then performs a CAS modifying the kth node. Thus, each node is modified by only one thread, but the modification is read by all other threads. This behavior is intended to represent the read-write synchronization of the DeleteMin operations. This corresponds to traversing the prefix of deleted nodes, then updating the first non-deleted node using CAS. Of the nodes in the deleted prefix, we can expect one modified node per other thread running, if we assume fair scheduling.

When n reaches 32 threads, the microbenchmark achieves roughly 2.6 million operations per second. This coincides roughly with the maximal throughput of the new algorithm in *Uniform*. The difference can be attributed to that, in the case of the microbenchmark, there is no false sharing at all, whereas in the actual implementation, some false sharing may be expected. We then conclude that the performance of the priority queue is entirely limited by the DeleteMin operation, and that, to achieve better performance, one would have to read less than one modified memory position per other thread and operation, on average.

7 Conclusion

We have presented a new linearizable, lock-free, skiplist-based priority queue algorithm, which achieves better performance than existing such algorithms, mainly due to reduced contention for shared memory locations. In addition, a simple benchmark indicates that the scalability of our algorithm is entirely limited by the logical deletion part of the DeleteMin operation. We believe that similar ideas for improved performance can be applied to other concurrent data structures, whose scalability is limited by contention for shared memory locations.

Acknowledgment The authors would like to thank Pavol Bauer and Stefan Engblom, whose work on the URDME framework inspired this work, and who provided and set up the URDME benchmark. We also thank Nikos Nikoleris for insightful discussions.

References

1. Crain, T., Gramoli, V., Raynal, M.: Brief announcement: a contention-friendly, non-blocking skip list. In: DISC. LNCS, vol. 7611, pp. 423–424. Springer (2012)
2. Drawert, B., Engblom, S., Hellander, A.: URDME : a modular framework for stochastic simulation of reaction-transport processes in complex geometries. BMC Systems Biology 6(76), 1–17 (2012)
3. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC'04. pp. 50–59. ACM (2004)
4. Fraser, K.: Practical lock freedom. Ph.D. thesis, University of Cambridge (2003)
5. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. LNCS, vol. 2180, pp. 300–314. Springer (2001)
6. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: SPAA. pp. 355–364. ACM (2010)
7. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
8. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc. (2008)
9. Holzmann, G.: The model checker SPIN. IEEE Trans. on Softw. Eng. SE-23(5), 279–295 (1997)
10. Hunt, G.C., Michael, M.M., Parthasarathy, S., Scott, M.L.: An efficient algorithm for concurrent priority queue heaps. Inf. Process. Lett. 60(3), 151–157 (1996)

11. Lindén, J., Jonsson, B.: A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In: Principles of Distributed Systems. LNCS, vol. 8304, pp. 206–220. Springer, Nice, France (Dec 2013)
12. Lotan, I., Shavit., N.: Skiplist-based concurrent priority queues. In: IPDPS. pp. 263–268. IEEE (2000)
13. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. 15(6), 491–504 (2004)
14. Molka, D., Hackenberg, D., Schone, R., Muller, M.: Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In: PACT. pp. 261–270. ACM (2009)
15. Pugh, W.: Concurrent maintenance of skip lists. Tech. Rep. CS-TR-2222, Dept. of Computer Science, University of Maryland, College Park (1990)
16. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (1990)
17. Sundell, H., Tsigas, P.: Scalable and lock-free concurrent dictionaries. In: SAC'04. pp. 1438–1445. ACM (2004)
18. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. 65(5), 609–627 (May 2005)
19. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: PODC'95. pp. 214–222. ACM (1995)
20. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Proc. of SPIN'09. LNCS, vol. 5578, pp. 261–278. Springer (2009)

A SPIN Model

In the SPIN model, we use the same approach as Vechev et al. [20]. A sequential priority queue model is managed in parallel with the concurrent model, and whenever the concurrent model reaches a linearization point of one of its operations, the result of the concurrent operation is compared to that of the sequential model.

The code in the following listing adheres to the old SPIN scope rules, to use it in SPIN version 6 or later, the `-O` flag has to be used. To The full model is also available through the homepage.

```
#define IF if ::
#define FI :: else fi

#define CAS(a, d, o, n) \
    cas_success = 0;     \
    if :: (d == 0 && a == o) -> a = n; cas_success = 1; \
    :: else fi

#define FAO(a,v) \
     a; a = v;

#define WHILE do ::
#define ELIHW :: else -> break; od

#define GCASSERT(new, old) \
    assert(nodes[new].recycled == 0 || nodes[old].recycled);

#define NLEVELS 3 /* 3 level skiplist */
#define THREADS 3 /* 3 threads */

#define MAX_KEY 10

#define MAX_OPS 2 /* no. of random ops per thread */
#define BOUNDOFFSET 2 /* restructure offset */

#define NODES 12  /* total memory */

/* Operation types. */
#define INS   0
#define DEL   1

/* types */
#define key_t byte
#define idx_t byte

typedef node_t {
  key_t key;
  byte level;
  bit inserting;
  bit recycled;
  /* the following 2 fields are colocated in one mem pos,
   * and should be treated as such. */
  bit d;
  idx_t next[NLEVELS];
}

typedef queue_t {
  idx_t head, tail;
}

/* this is the memory */
node_t nodes[NODES];

/**********  declaration of global variables *************/

queue_t q; /* the priority queue */
byte seqq[NODES]; /* the sequential spec. */
idx_t glob_entry; /* pointer to free memory */

/********* sequential specification **************/

/* adding */
inline seq_add(entry, k) {
  assert(seqq[k] == 0);
  seqq[k] = 1;
}

/* removing - element should be the smallest */
inline seq_remove(kl) {
  assert(seqq[kl]);
    for (j : 0..kl-1) {
    assert(seqq[j] == 0);
  }
  seqq[kl] = seqq[kl] - 1;
}
/* if empty, no entry in queue */
inline seq_empty() {
  for (j : 0..(NODES-1)) {
    assert(seqq[j] == 0);
  }
}

/************* Handling nodes/memory *****************/

inline get_entry(ptr)
{
  d_step{
    ptr = glob_entry;
    assert(ptr < NODES - 1);
    glob_entry++;
  }
}

/* return index pointing to a node being free to use */
inline alloc_node(new, k)
{
  atomic {
    get_entry(new);
    nodes[new].key = k;
    select(i : 0..(NLEVELS - 1)); /* ok, called before locatepreds */
    nodes[new].level = i;
    nodes[new].inserting = 1;
  }
}


/******************************************************************
 * BEGIN PRIORITY QUEUE ALGORITHM
 ******************************************************************/


/* CAS(addr, d, old, new) - representing a CAS, that will update addr
 * to new, given that addr = old and d = 0. d represents hence the
 * delete bit being a part of old. */

/* FAO(addr, val) - representing a Fetch-and-Or, that will update
 * addr to *addr | val. */

inline LocatePreds(key) {
  d_step { /* resetting some local vars */
    cur = 0; pred = 0; d = 0; del = 0;
    i = NLEVELS; pred = q.head
  }
  /* NB: index i is offset by one in comparison to paper,
   * due to lack of negative bytes in promela */
  WHILE (i > 0) ->  /* for each level */
    d_step { /* colocated together */
      cur = nodes[pred].next[i-1];
      d = nodes[pred].d
    }
    WHILE (nodes[cur].key < key || nodes[cur].d || (d && i == 1)) ->
      atomic {
        IF (d && i == 1) -> del = cur FI;
        pred = cur; /* local */
        /* colocated together */
        cur = nodes[pred].next[i-1];
        d = nodes[pred].d
      }
    ELIHW;
    atomic { /* local vars */
      preds[i-1] = pred;
      succs[i-1] = cur;
      i-- /* descend to next level */
    }
  ELIHW
}

inline Insert(key) {
  alloc_node(new, key)

retry:
  LocatePreds(key)

  nodes[new].next[0] = succs[0];
  /* Lowest level */
  atomic { /* linearization point of non-failed insert */
    CAS(nodes[preds[0]].next[0], nodes[preds[0]].d, succs[0], new);
    if :: (cas_success) ->
          seq_add(new, key)
          GCASSERT(succs[0], new)
       :: else -> goto retry /* restart */
    fi
  }
  /* swing upper levels */
  j = 1; /* i is being used in locatepreds */
  WHILE (j <= nodes[new].level) ->
    nodes[new].next[j] = succs[j];
    IF (nodes[new].d || nodes[succs[i]].d || succs[i] == del) ->
       goto end_insert
    FI;
    atomic {
      CAS(nodes[preds[j]].next[j], 0, succs[j], new);
      IF (cas_success) ->
        GCASSERT(succs[j], new)
        j++
      FI
    }
    IF (!cas_success) ->
      LocatePreds(key) /* update preds, succs and del */
      IF (succs[0] != new) -> goto end_insert FI
    FI
  ELIHW;
end_insert:
  nodes[new].inserting = 0
}

inline Restructure() {
  i = NLEVELS - 1; pred = q.head;
```

```
re_continue:
  WHILE (i > 0) ->
    h = nodes[q.head].next[i];
    cur = nodes[pred].next[i];
    IF (!nodes[h].d) -> i--; goto re_continue FI;
    WHILE (nodes[cur].d) ->
      pred = cur;
      cur = nodes[pred].next[i]
    ELIHW;
    atomic {
      CAS(nodes[q.head].next[i], 0, h, nodes[pred].next[i]);
      IF (cas_success) ->
        GCASSERT(nodes[pred].next[i], q.head)
        i--
      FI
    }
  ELIHW
}

inline DeleteMin () {
  d_step {
    d = 1; x = q.head; offset = 0;
    obshead = nodes[x].next[0]
  }
  WHILE (d) ->
    atomic {
      offset ++;
      /* nxt & d colocated */
      nxt = nodes[x].next[0];
      d   = nodes[x].d;
      IF (nxt == q.tail) ->
        /* empty: got linearized when reading nxt */
        seq_empty()
        goto end_remove
      FI
    }
    IF (nodes[x].inserting && newhead == NODES) ->
      newhead = x
    FI;
    atomic {
      /* linearization point */
      d = FAO(nodes[x].d, 1)
      IF (!d) ->
        /* check linearization */
        key = nodes[nodes[x].next[0]].key;
        seq_remove(key)
      FI
    }
    x = nodes[x].next[0]
  ELIHW;
  IF (offset <= BOUNDOFFSET) -> goto end_remove FI;
  IF (newhead == NODES) -> newhead = x FI;
  atomic {
    CAS(nodes[q.head].next[0], 0, obshead,newhead);
    if :: (cas_success) -> GCASSERT(newhead, q.head)
       :: else -> goto end_remove
    fi
  }
  Restructure()
  cur = obshead;
  WHILE (cur != newhead) ->
       nxt = nodes[cur].next[0];
       nodes[cur].recycled = 1; /* MarkRecycle */
       cur = nxt
  ELIHW;
end_remove:
}


/*******************************************************************
 * END ALGORITHM
 *******************************************************************/


/* Random key generator that generates unique keys
 * 0 is taken by head sentinel node
 * MAX_KEY is taken by tail sentinel node, and should be > keys[*] */

bit keys[MAX_KEY] = 1;

inline pick_key(var) {
    atomic {
    if :: (keys[1] == 1) -> keys[1] = 0; var = 1
       :: (keys[2] == 1) -> keys[2] = 0; var = 2
       :: (keys[3] == 1) -> keys[3] = 0; var = 3
       :: (keys[4] == 1) -> keys[4] = 0; var = 4
       :: (keys[5] == 1) -> keys[5] = 0; var = 5
       :: (keys[6] == 1) -> keys[6] = 0; var = 6
       :: (keys[7] == 1) -> keys[7] = 0; var = 7
       :: (keys[8] == 1) -> keys[8] = 0; var = 8
       :: (keys[9] == 1) -> keys[9] = 0; var = 9
    fi;
    }
}

inline start_op() {
  init_locals();
};

inline end_op() {
  d_step {
  key = 0;
  op = 0;
  new = 0;
  }
}

/* randomly choose a deletemin or an insert operation */
inline exec_op(key) {
  start_op();
  assert(key < NODES);
  if
    :: op = INS;
       pick_key(key);
       Insert (key);
    :: op = DEL;
       DeleteMin();
  fi;
  end_op();
}

/* a thread execution is a sequence of MAX_OPS operations */
inline execute()
{
  byte _dummy1;
  for (_dummy1 : 1..(MAX_OPS)) {
    exec_op(key);
  }
}

/* initialize local variables */
inline init_locals()
{
  d_step {
    pred = 0;
    cur = 0;
    d = 0;
    preds[0] = 0;
    preds[1] = 0;
    preds[2] = 0;
    succs[0] = 0;
    succs[1] = 0;
    succs[2] = 0;
    op = 0;
    offset = 0;
    obshead = 0;
    del = 0; /* ok, succs will never be 0 */
    cas_success = 0;
    h = 0;
    i = 0;
    j = 0;
    new = 0;
    key = 0;
    x = 0;
    nxt = 0;
    newhead = NODES;
  }
}

/* declare local variables */
inline define_locals()
{
  idx_t pred, cur, obshead, offset, newhead, h, x, nxt;
  idx_t preds[NLEVELS], succs[NLEVELS], del;
  byte i,j;
  bit op, d, cas_success;
  byte key;

  idx_t new;
  init_locals();
}

/* a thread initializes local variables, and
 * randomly executes operations */
proctype client() {
  define_locals();
  execute();
}

 /* global initialization of the structure */
inline init_globals()
{
  atomic {
    glob_entry = 0;
    /* tail */
    alloc_node(new, MAX_KEY);
    q.tail = new;
    nodes[q.tail].level = 1;
    nodes[q.tail].inserting = 0;

    alloc_node(new, 0);
    q.head = new;
    nodes[q.head].level = 1;
    nodes[q.head].inserting = 0;
    for (j : 0..2) { /* levels */
      nodes[q.head].next[j] = q.tail;
    };
  }
}
```

```
/* entry point */

init {
  atomic{
    byte _dummy0;
    define_locals();
    init_globals();
    /* run n - 1 threads as proctype */
for ( _dummy0 : 1..(THREADS - 1)) {
      run client();
    }
  }
  /* and run last thread here */
  execute();

  /* wait until the other process finishes. */
  _nr_pr == 1;
}
```