A Conflict-Resilient Lock-Free Linearizable Calendar Queue[*]

ROMOLO MAROTTA, Tor Vergata University of Rome, Italy
MAURO IANNI, Lockless S.r.l., Italy
ALESSANDRO PELLEGRINI, Tor Vergata University of Rome, Italy
FRANCESCO QUAGLIA, Tor Vergata University of Rome, Italy

In the last two decades, great attention has been devoted to the design of non-blocking and linearizable data structures, which enable exploiting the scaled-up degree of parallelism in off-the-shelf shared-memory multi-core machines. In this context, priority queues are highly challenging. Indeed, concurrent attempts to extract the highest-priority item are prone to create detrimental thread conflicts that lead to abort/retry of the operations. In this article, we present the first priority queue that jointly provides: i) lock-freedom and linearizability; ii) conflict resiliency against concurrent extractions; iii) adaptiveness to different contention profiles; and iv) amortized constant-time access for both insertions and extractions. Beyond presenting our solution, we also provide proof of its correctness based on an assertional approach. Also, we present an experimental study on a 64-CPU machine, showing that our proposal provides performance improvements over state-of-the-art non-blocking priority queues.

CCS Concepts: • **Theory of computation** → **Concurrent algorithms**; • **Information systems** → **Message queues**; • **Computing methodologies** → *Discrete-event simulation*; • **Software and its engineering** → *Synchronization.*

Additional Key Words and Phrases: Data structures design and analysis, Shared memory algorithms, Concurrent algorithms, Non-blocking Priority queue, Pending Event Set

ACM Reference Format:
Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2023. A Conflict-Resilient Lock-Free Linearizable Calendar Queue. *ACM Trans. Parallel Comput.* 1, 1, Article 1 (January 2023), 33 pages. https://doi.org/10.1145/3635163

1 INTRODUCTION

The Pending Event Set (PES) is a building block for event-driven applications, whose computation advances thanks to the occurrence of timestamped events. In these applications, events are inserted in a PES—also denoted as *event pool*—as soon as they are scheduled to occur. They are subsequently extracted and processed in an order that depends on the associated timestamps. For this reason, this data structure is often implemented as a priority queue, where timestamps determine priorities. When the event-driven paradigm merges with concurrent and parallel computing, effective concurrent management of the event pool is mandatory to provide scalability [28, 30].

The extensive literature on event-driven applications (like discrete event simulation) proposes several event pool data structures with (amortized) constant-time access, such as classical Calendar Queues [4], Ladder Queues [33] and LOCT [27]. However, these solutions are *non-concurrent*, meaning that their usage in concurrent applications requires threads to access the data structure

[*] This is a revised and extended version of a paper presented at the ACM SIGSIM PADS 2017 conference [25].

Authors' addresses: Romolo Marotta, romolo.marotta@gmail.com, Tor Vergata University of Rome, Rome, Italy; Mauro Ianni, ianni@lockless.it, Lockless S.r.l., Rome, Italy; Alessandro Pellegrini, a.pellegrini@ing.uniroma2.it, Tor Vergata University of Rome, Rome, Italy; Francesco Quaglia, francesco.quaglia@uniroma2.it, Tor Vergata University of Rome, Rome, Italy.

in a lock-protected critical section guaranteeing isolation. Clearly, this is a scalability limitation, despite the advantageous time complexity offered for every single operation on the pool. These solutions are mainly based on: (i) a multi-dimensional arrangement (e.g., multi lists) of events where the number of events per dimension is bounded by a constant; (ii) the presence of a reference always pointing (near) to the maximum priority event (i.e., the one with the minimum timestamp); (iii) an arrangement of the data structure that can be partially or totally reshuffled to control the number of events in each dimension.

Recently, concurrent management of the event pool has gained interest to enable its effective exploitation in shared-memory multi-core architectures [28]. One core point along this direction has been the exploitation of non-blocking synchronization [16, 18] applied to priority queues [22, 32]. However, none of the recent approaches—like Lock-free Ladder Queues [13] and Splay Trees [1]—offers a PES that jointly ensures lock freedom, constant-time access and is proved linearizable. On the other hand, providing concurrent (most notably non-blocking), scalable, and linearizable priority queues is challenging because of the difficulties in handling the event extraction with the minimum timestamp. Intuitively, the minimum timestamp event in the PES is materialized at only one item at any time. Consequently, multiple concurrent extractions cannot succeed at the very same time. This impacts particularly non-blocking concurrent algorithms, where no isolation is exploited in the access by threads to the data structure. These algorithms are based on performing work not visible to other threads—such as reading a few parts of the data structure—and executing an atomic machine instruction (e.g., `Compare&Swap`), which can fail while attempting to perform some update. The failure leads to the abort of the operation (e.g., the extraction of the minimum), which needs to be retried in the hope not to lose again in any (possible) conflict with other concurrent thread operations.

We also note that non-blocking PES solutions significantly stress the cache subsystem since the memory region near the minimum—namely, the data it is represented by—can be frequently exchanged across threads (in terms of the corresponding cache lines' state in the cache coherency protocol) also because of retried operations. A few approaches [22, 25] face this issue by trading off latency in favour of throughput, with the limitation that the contention level (the number of threads to be managed) has to be known a priori. Nowadays, satisfying this requirement is very challenging, especially when considering dynamic workloads or virtualized hardware. Other solutions try to completely overcome the scalability bottleneck of priority queues by renouncing to provide linearizability [1, 13].

In this work, we present a lock-free, conflict-resilient and contention-adaptive calendar queue that is also provably linearizable, which overcomes all the limitations of the results currently available in the literature. To prove the correctness of our algorithm, we followed an assertional approach, which offers the reader an alternative perspective compared to the well-known approach based on the exploitation of fixed-linearization points in the algorithm. Indeed, our algorithm is based on procedures that do not always linearize at the same machine instructions—hence there is no a priori identification of what machine instruction will need to be considered as the linearization point of a method invocation (insertion/extraction)—just depending on what happens in terms of concurrency of the operations by multiple threads. Beyond presenting our algorithm and its correctness proof, we also report the results of an experimental study comparing it to literature references on a 64-CPU machine, showing that our proposal has relevant, pragmatic benefits in terms of scalability and operations throughput. This study has been based on the well-known Hold benchmark [29], commonly exploited in the literature to assess the effectiveness of PES algorithms.

The remainder of this article is structured as follows. Related work is discussed in Section 2. The proposed solution is presented in Section 3. Section 4 and Section 5 give the proof of liveness (lock freedom) and safety (linearizability) of our data structure, respectively. The experimental study is presented in Section 6. Section 7 concludes the article.

2 RELATED WORK

The original Calendar Queue [4] is a timestamp-ordered data structure based on multi-lists, which offers amortized constant time insertion of events with generic timestamps and constant-time extraction of the event with the minimum timestamp. The Ladder Queue [33] is a variant of the Calendar Queue which is more suited for skewed distributions of the timestamps of the events, thanks to the possibility to split an individual bucket into sub-intervals dynamically (i.e., sublists of records) when the number of elements associated with the bucket exceeds a given threshold. LOCT [27] is an additional variant that allows reducing the actual overhead for constant-time insertion/extraction operations thanks to the introduction of a compact hierarchical bitmap indicating the status of any bucket (empty or not). None of these proposals has been devised for concurrent accesses. Therefore, their usage by concurrent threads would require locks for serializing the accesses, which can be detrimental to scalability, as shown in [13].

The work in [2] provides an event-pool data structure enabling parallel accesses via fine-grain locking of a sub-portion of the data structure upon performing an operation. However, the intrinsic scalability limitations of locking still make this proposal unsuitable for significant levels of parallelism, as also shown in [29].

As for lock-free management of sets by concurrent threads, various proposals exist (e.g., lock-free linked lists [14] or skip lists [22, 32]), which anyhow do not offer constant-time operations. The lock-free linked list pays a linear cost for ordered insertions. Skip lists [26] are randomized multi-linked lists that provide logarithmic access time with guarantees similar to a randomized binary search tree [9].

Fraser [12] provided a practical lock-free and linearizable implementation of a set of elements based on a skip list by applying ideas similar to the Harris' lock-free linked list—namely logical deletion by node marking. Based on this work, other skip lists have been designed to address specific subproblems, such as cache locality [7, 10] and Non-Uniform-Memory-Access (NUMA) [8].

Lotan and Shavit [31] employ this data structure as a concurrent priority queue. The original implementation was not linearizable since extracting an item inserted after the enqueue of a new minimum was possible. Herlihy and Shavit [17] added a timestamping mechanism to overcome this problem and achieve linearizability. Sundell and Tsigas [32] proposed a lock-free skip-list-based implementation that uses the marking strategy for dequeued items. Linearizability is obtained by imposing that physical removal immediately follows logical deletion. Unfortunately, since threads continuously compete for memory locations of items close to the minimum, all the above implementations suffer from very high contention under standard workloads. Lindén and Jonsson [22] presented an elegant design to reduce the bottleneck of deleting the minimal element. The latter is considered the state-of-art implementation of lock-free linearizable priority queues [1] but still does not offer constant-time operations. Another solution based on skip lists is presented in [1], where the authors propose a priority queue with a relaxed semantics, which allows to retrieve non-minimum items in favor of an improved scalability.

Lock-free operations in combination with constant-time complexity have been studied in [13], which presents a variation of the Ladder Queue where the elements are at any time bound to the correct bucket, but the bucket list is not ordered. Constant-time is achieved since the extraction from an unordered bucket returns the first available element, which does not necessarily correspond to the one with the minimum timestamp. At the same time, linearizability is not supported. This proposal can be exploited in applications relying on speculative processing, where unordered extractions leading to causal inconsistencies are reversed via proper rollback mechanisms—this is the case of speculative Parallel Discrete Event Simulation (PDES). However, a few recent results [6, 20, 30] have shown that linearizable fetching of events from the shared pool still represents the core solution, even when exploiting speculation. The correct order of delivery is guaranteed by the algorithm we present in this article since we always deliver the highest priority event currently in the event pool, which has been inserted by any operation that is linearized before the extraction.

The recent proposal in [15] explores the idea of managing concurrent accesses to a shared pool by relying on Hardware Transactional Memory (HTM). Insertions and extractions are performed as HTM-based transactions, hence in non-blocking mode. However, HTM-based transactions can abort for several reasons, not necessarily related to conflicting concurrent accesses to the same portion of the data structure. For example, they can abort because of false cache sharing or limited cache capacity, which might be adverse to executions with very large event pools. The algorithm we present in this article does not require special hardware support, thus entirely eliminating the secondary effects caused by HTM limitations on the abort rate of the operations.

The proposal in [24] is based on lock-free access to a multi-bucket data structure and provides amortized $O(1)$ time complexity for both insertion and extraction operations. However, it does not provide a lock-free scheme for the dynamical resize of the bucket width. Hence, to achieve adequate amortizing factors, all the threads would need to (periodically) synchronize via locks to change the bucket width and redistribute events over the reshaped buckets. On the other hand, avoiding at all the synchronized reshuffle of the buckets might give rise to non-competitive amortizing factors (e.g., too many elements associated with a bucket). This problem is avoided at all by the algorithm we present in this article.

Finally, the work in [23] enables a non-blocking reshuffle of the data structure but does not guarantee conflict resilience of extraction operations targeting the "hot" bucket to which the locality of the extraction activities is bound. Hence, as soon as two or more extraction operations are executed concurrently and conflict, just one of them is allowed to be finalized with no retry cycle. In this article we overcome these issues at all by providing resilience to conflicting extractions. As we also show in the experimental study, this feature of our algorithm provides clear advantages in terms of scalability and performance.

As an additional limitation of these recent proposals (i.e., [23, 24]), linearizability is not proven to be supported. Instead, the lock-free algorithm we present in this article is formally proven linearizable.

3 THE LOCK-FREE CALENDAR QUEUE

Our proposal, called Lock-Free Calendar Queue (LFCQ), is a non-blocking priority queue tailored to maintain the PES, directly inspired by the classical Calendar Queue [4]. For this reason, we often use the words item/event and priority/timestamp interchangeably.

In the Calendar Queue, the priority domain is split into equal partitions, called *time slots*, each one covering a given range of priorities, whose length is called *bucket width* (bw). Each time slot is served by a dedicated priority queue called *virtual bucket* (VB). Since the priority domain is an infinite set of real numbers, the number of VBs is unbounded, and, thus, they are mapped circularly to a finite array of *physical buckets* (PBs).

Essentially, a Calendar Queue is a bi-dimensional data structure with an array of lists (the PBs), each maintaining items of multiple VBs. When a new event e with timestamp T_e is enqueued, the index I_{vb} of the VB associated with the time slot of e is computed as $I_{vb} = \lfloor T_e/bw \rfloor$. At this point, the index I_{pb} of the target PB is computed as $I_{pb} = (I_{vb} \mod L)$, where L is the calendar length, namely the number of PBs. Finally, e is inserted into the I_{pb}-th physical bucket.

An integer C stores the index of the VB containing the event with the minimum timestamp, namely the one with the highest priority. A dequeue operation starts by retrieving the value C to

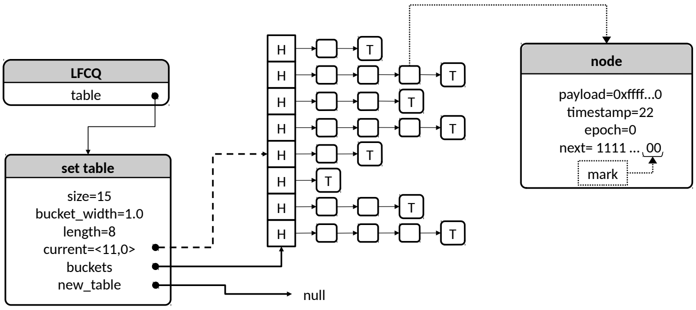


Fig. 1. Scheme of the LFCQ data structure.

extract the minimal key stored into the C-th VB, which is actually stored into the C_{pb}-th physical bucket, where $C_{pb} = (C \mod L)$. When the VB identified by C is empty, the operation looks for an event to extract from the next bucket, thus also performing the update of C by one unit.

The Calendar Queue specification achieves amortized constant-time access for insertion and extraction operations. This is guaranteed by reshuffle operations that keep the number of items in a PB bounded by a constant. In particular, when the number of events halves or doubles, the data structure is entirely reshuffled by adopting a new bw that changes the partitioning of the priority domain and, consequently, the mapping between events and buckets.

We recall again that the Calendar Queue is a sequential algorithm, not allowing concurrent operations by threads. Its usage in applications with multiple concurrent threads requires the inclusion of mechanisms—like locking—for the isolated execution of its operations.

Key idea. In principle, our LFCQ algorithm implements the same logic as the Calendar Queue but provides non-blocking progress, particularly lock freedom. Here, the main challenge is guaranteeing linearizability while jointly providing: 1) lock-free management of physical buckets; 2) lock-free management of the index pointing to the virtual bucket containing the minimum; 3) non-blocking reshuffle phases.

For point 1), our algorithm exploits the well-known *non-blocking set* by Harris [14], which we have augmented with additional capabilities required to provide the features in points 2) and 3). In particular, we resort to the notion of logical timestamps, called *epochs*, but with a different flavour than just providing correctness, as instead done in [17]. In fact, epochs have a dual role in our LFCQ algorithm. On the one hand, they ensure both correctness *and* conflict resilience of extractions within a single virtual bucket. On the other hand, epochs jointly allow a consistent view of multiple priority queues (buckets) and act as if they were a single connected queue. Finally, in LFCQ, we guarantee that active insertion and deletion operations cannot update critical points of the data structure during the reshuffle phases. This allows items to be moved across calendars without hampering the consistency of the data structure. To this goal, the state machine representing the evolution in time of nodes within the non-blocking list has been augmented with additional states and transitions for event migration that are embedded into both the records keeping individual events and metadata representing the current state of a physical bucket.

3.1 LFCQ Baseline Organization

The basic organization of our LFCQ is provided in Figure 1. It consists of a pointer `table` to a data structure called *Set Table*, which maintains the metadata required for queue management. In particular, `bucket_width` stores the actual bw currently used to partition the priority domain;

`length` is the number of PBs in the calendar; `size` keeps the number of stored events; `current` is a pair $\langle index, epoch\rangle$ such that *index* logically points to the VB containing the minimum and *epoch* is a counter which is incremented each time insertion in the past of the current minimum occurs (a new minimum is inserted); `buckets` is a pointer to the array of PBs, that are implemented as non-blocking linked lists; `new_table` is a reference initialized to `null`, which is used only during reshuffle phases.

To ensure consistency with concurrent accesses, the queue operations rely on the following 64-bit atomic RMW [21] instructions: `Fetch&Add`(a,b) (`FAD`) atomically increments the value of a memory cell a with a supplied value b and returns its value before the update; `Compare&Swap`(a,b,c) (`CAS`) atomically updates a given memory location a with a value c only if its current content is equal to a given value b, otherwise no memory update takes place and we say that the `CAS` instruction fails.

3.1.1 *Physical Bucket Queues.*

While PBs in the original non-concurrent Calendar Queue [4] are priority queues implemented as ordered linked lists, our algorithm is based on a different specification of PBs called Physical Bucket Queue (PBQ), which is a priority queue with a slightly modified semantics. First, its enqueue and dequeue APIs, respectively called *Connect* and *GetMin*, take two additional parameters: a VB index and an integer value called epoch. On the one hand, the CONNECT procedure adds a new item to the queue and creates an association between the item and the epoch passed as a parameter. This is kept into the `epoch` field of a node encapsulating the new item (see Figure 1). On the other hand, the GETMIN procedure tries to extract the highest priority item belonging to the VB passed as a parameter, which is compatible with the epoch value provided as input. If there are no items in the PBQ, it returns the special value $\emptyset$. Conversely, if the PBQ is not empty, but no item belongs to the interested VB, it returns $\perp$. Finally, if the target VB is not empty, the result depends on the epoch e_i associated with the highest priority item i. If e_i is lower than or equal to the epoch value passed as a parameter, GETMIN extracts and returns i; otherwise, it returns a special value `ABORT` indicating that the procedure cannot succeed due to some conflicting concurrent operation.

The second main difference between a PBQ and a priority queue stands in the additional APIs for migrating items from a PBQ to another one in a non-blocking fashion. These APIs are exploited during the non-blocking reshuffling of the data structure. In particular, BLOCK atomically puts a PBQ in a state that does not allow the insertion of a new minimum-timestamped event; BLOCKNEXT atomically puts a PBQ and the successor e' of the event e passed as a parameter in a state that does not allow the extraction of e' and the insertion of a timestamp between the one of e and e'; MIGRATE atomically removes and inserts an item from one PBQ to another one. Clearly, both CONNECT and GETMIN can return the `ABORT` value when they try to operate on elements whose state "blocks" extractions/insertions (mainly, during reshuffle operations).

Nodes of Physical Bucket Queues. Our PBQs are mainly based on the non-blocking set implementation by Harris [14]. Nodes within this data structure have two possible states: *valid* and *logically deleted*. The former signals that the node contains a key that can be safely extracted. Conversely, the latter indicates that the key has been already extracted, but the node is still connected to the linked list and must be physically removed. This is because Harris' linked list exploits lazy deletion to provide correct concurrent management of the set (e.g., correct concurrent traversal while deleting nodes). To this end, it embeds the state of a node into the Least Significant Bit (LSB) of the field, maintaining the pointer to the next node in the list.

Unlike other priority queues based on the Harris' list, our PBQ needs two additional states to correctly handle migrations of nodes from one queue to another one. Consequently, we need to steal an extra bit from the field `next` of a node (see Figure 1). In particular, we have four states of a

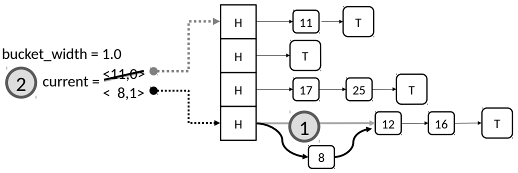


Fig. 2. Enqueue main steps.

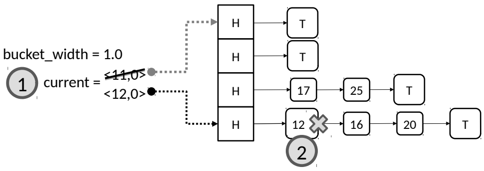


Fig. 3. Dequeue main steps.

node, each one mapped to different values of the two LSBs, called *mark*, in the field used to point to a subsequent node:

- `VAL`: the node stores a valid key, which corresponds to the node timestamp (the value of the mark field is 00);
- `DEL`: the node stores a key that cannot be extracted (01) since the node encapsulating it is no longer valid, namely the corresponding key was already extracted or migrated to a new PBQ;
- `MOV`: the node stores a key that cannot be extracted since it is going to be migrated in a new PBQ (11);
- `INV`: the node stores a key that might be valid in the future (10).

During the extraction, the GetMin procedure applies the transition from `VAL` to `DEL` as in the original non-blocking linked list by Harris. Other states are reachable only during, or due to, reshuffle phases. Consequently, these transitions will be discussed deeply in Section 3.4 where details of the reshuffle protocol are given. Both `DEL` and `INV` can be considered as logically deleted nodes and, consequently, they are disconnected while searching for a specific key. However, there are scenarios related to the reshuffle phase that require only `DEL` nodes to be actually disconnected. For this reason, the Search routine takes an additional parameter indicating which states must be considered for physically removing nodes.

As a last note, we will present LFCQ under the assumption that all the timestamps of different events have different values. This can be achieved in practice by having a timestamp representation based on a couple where one field is the actual timestamp, and the other field is a unique identifier taken, for example, by an atomic counter.

3.2 Enqueue Operation

The Enqueue() procedure of the LFCQ, whose pseudocode is shown in Algorithm 1, has three main phases. First, it checks if a reshuffle phase has begun (by evaluating the `new_table` field) or a new one has to start (line E5). In the positive case, it joins the refactoring protocol and, once completed, restarts from the beginning. Once the current set table is obtained, it proceeds by inserting the event e (received as a parameter) with priority (namely, timestamp) $e.\texttt{t}$. To this goal, it determines the indexes vb and I_{pb} of the virtual and physical buckets associated with the priority $e.\texttt{t}$ (lines E8-E10). At this point, it reads `current`, obtaining the actual epoch which is written into the new node and tries to add e to the I_{pb}-th PBQ (denoted as pb) by invoking the Connect API. Upon the Connect invocation, if the timestamp to be inserted is already present, the operation fails to notify that no new node is inserted, and the returned value $\perp$ indicates the violation with respect to timestamp uniqueness. If the timestamp is not already present, it checks if the inserted item belongs to a VB preceding the one pointed by `current`. When $vb \leq \texttt{current.index}$, the enqueue updates the index kept by `current` (an item has been inserted in the past of the previous minimum timestamp) and the epoch of `current` is incremented by one. In order to guarantee that the insertion becomes visible to extractions after its completion, this step is performed inside a retry loop (lines E15-E20) with a `CAS`. In this way, a new phase of the queue, where all the new items will be associated with

Algorithm 1 lock-free Enqueue

```
procedure Enqueue(event e)
    $res \gets$ ABORT
    while $res \neq \top$ do
        $h \gets$ table                                  ▷ Obtain table
        if checkResize($h$) $\vee$ $h$.new_table $\neq$ NULL then
            Reshuffle()                         ▷ Join the reshuffle protocol
            continue
        $vb \gets \lfloor e.t/h.bw \rfloor$             ▷ Compute virtual bucket
        $I_{pb} \gets vb \mod h$.length
        $pb \gets h$.buckets[$I_{pb}$]                     ▷ Obtain PBQ
        $ep \gets h$.current.epoch            ▷ Obtain current epoch
        $res \gets pb$.Connect($vb$, $e$, $ep$)      ▷ Try to add e to pb
        if $res = \perp$ then
            return $\perp$
    while $\top$ do
        $old \gets h$.current
        $oldIn \gets old$.index
        $oldEp \gets old$.epoch
        $new \gets \langle vb, oldEp + 1 \rangle$
        if $vb > oldIn \vee$ CAS(&$h$.current, $old$, $new$) then
            break
    Fetch&Add(&$h$.size, 1)
    return $\top$
```

```
procedure Connect(integer $vb$, event $e$, integer $ep$)
    if RandomClean() then                 ▷ Trigger housekeeping task
        $\langle l, l\_next, r \rangle \gets$ Search(0, DEL $\vee$ INV)
        CAS($l$.next, $l\_next$, $e$)
    $\langle l, l\_next, r \rangle \gets$ Search(e.t, DEL $\vee$ INV)
    if $l$.t = $e$.t then                            ▷ Key already present
        return $\perp$
    $e$.epoch $\gets ep$
    $e$.next $\gets r$
    if $\neg$CAS(&$l$.next, unmark($l\_next$), $e$) then
        return ABORT                                 ▷ Failed insertion
    return $\top$
```

Algorithm 2 lock-free Dequeue

```
procedure Dequeue()
    $iter \gets 0$
    while $true$ do
        $h \gets$ table                                  ▷ Obtain table
        if checkResize($h$) $\vee$ $h$.new_table $\neq$ NULL then
            Reshuffle()                         ▷ Join the reshuffle protocol
            continue
        $cur \gets h$.current                          ▷ Obtain current
        $epoch \gets cur$.epoch                   ▷ Obtain current epoch
        $vb \gets cur$.index                   ▷ Compute virtual bucket
        $newC \gets \langle vb + 1, epoch \rangle$
        $I_{pb} \gets vb \mod h$.length
        $pb \gets h$.table[$I_{pb}$]                      ▷ Obtain P-bucket
        $res \gets pb$.GetMin($vb$, $epoch$)
        if $res = \emptyset \wedge h$.length = 1 then
            return $\perp$
        if $res = \perp \vee res = \emptyset$ then
            CAS(&$h$.current, $cur$, $newC$)
            $iter$++
            if $iter >$ TH then
                Reshuffle()
                $iter \gets 0$
        else if $res \neq$ ABORT then
            Fetch&Add(&$h$.size, -1)
            return $value$
```

```
procedure GetMin(integer $vb$, integer $ep$)
    $cur \gets$ head
    while $cur$.ts $\leq$ bucket_width$\cdot vb$ do
        $c\_next \gets cur$.next
        if $cur$.epoch $> ep \vee$ isMarked($c\_next$, MOV) then
            return ABORT
        if $\neg$isMarked($c\_next$, VAL) then
            $cur \gets$ unmark($c\_next$)
            continue
        if $c$.ts $<$ bucket_width$\cdot vb \vee cur$ == head then
            continue
        if CAS(&$cur$.next, $c\_next$,
                        mark($c\_next$, DEL)) then
            return $cur$.payload
    if isMarked(head.next, MOV) then
        return ABORT
    if $cur$ = tail then
        return $\emptyset$
    return $\perp$
```

the new epoch, begins. These two main steps (insertion and current update) are depicted in Figure 2. Finally, as the last operation, the size field is incremented by one via an atomic FAD to reflect that the pool maintains an additional item.

The internals of Connect, shown in Algorithm 1, are quite straightforward. It invokes the search procedure that scans the linked list looking for the nodes that should surround the new item. The left node l is such that $l.t \leq e.t$. If both e and l actually have the same key, the Connect and, hence, the Enqueue fail. Conversely, if they store a different key, the Connect tries to insert the new event with a CAS. The CAS might fail not only because of concurrent insertions or extractions but also due to the left node being marked for migration (as MOV) in the meanwhile. As a last note, the insertion is used to disconnect items that are no longer needed, i.e., DEL and INV nodes.

3.3 Dequeue Operation

The main steps of the Dequeue in LFCQ are shown in Figure 3 and the pseudocode is given in Algorithm 2. Similarly to the enqueue operation, a dequeue starts by checking if a reshuffle has begun and retrieving a valid table reference (lines D4-D7). Then, it fetches the `current` field to extract the *index* of the VB, storing the minimum-timestamp event and the current *epoch*. At this point, it invokes the GetMin routine to fetch and remove the highest priority item. If the GetMin returns $\emptyset$, the PBQ has been detected as empty, and there is no reshuffle involving the current PBQ. In this case, it checks if the whole queue is empty. If the set table size is equal to one and the unique PBQ in the queue is empty, the Dequeue operation terminates by returning $\perp$ (line D15). Conversely, if the table size is greater than one or GetMin has returned $\perp$, the PBQ has no items belonging to the current VB. Consequently, before restarting from scratch, the Dequeue increases the value field of `current` and a local counter *iter* (see lines D18-D19 and step 1 of Figure 3). If GetMin returns `ABORT` or *iter* has passed a given threshold—the latter is required to ensure that eventually an always-empty queue will be represented with a set table of size equal to 1—a reshuffle will be triggered. Finally, if it succeeds, the `size` field is decremented by one, and it finally completes (lines D23-D25).

The internals of GetMin are shown in Algorithm 2. It scans the non-blocking linked list by looking for a candidate for extraction, namely a valid node with both timestamp and epoch compatible with the ones passed as parameters. If during this scan a node either marked as `MOV` or with an epoch greater than the one passed as parameter has been detected, it returns `ABORT` (see lines G5-G6). Conversely, non-valid nodes or nodes with a too small timestamp are simply skipped (see lines G10-G11). Once a candidate is found, GetMin tries to extract the item by marking it as `DEL`. If no item matches the search criteria, the last reached node is analyzed. If it is a `tail` node, it means that the end of the linked list has been reached, so the PBQ can be considered empty. In this case, GetMin returns $\emptyset$; otherwise, it means that some item belonging to VBs in the past or future of the target VB exists. Hence it returns $\perp$.

3.4 Resizing the Queue

The pseudocode of the Reshuffle operation of LFCQ is shown in Algorithm 3. $O(1)$ amortized time complexity of insertion/extraction operations is guaranteed because, on average, the number of elements within each PB is balanced. As said before, every time an operation on the queue is performed, a Reshuffle procedure might be called to exchange the current set table with a new one. This happens whenever the number of elements in the queue is unbalanced with the number of PBQs. In particular, the resize is executed if `size` oversteps a certain threshold (see the checkSize() and newSize() procedures in Algorithm 3). During the resize algorithm, it is ensured that the current set table is somehow frozen in a way that extractions and insertions cannot complete. Once this freezing stage is completed, we start migrating items from the old set table to the new one. Whenever a thread detects that the current set-table is (or is going to be) frozen, it starts collaborating in the migration process.

The first step of Reshuffle consists in announcing the beginning of this protocol by setting the `new_table` field of the old set table to refer to a new (just allocated) set table (line R3). This somehow “freezes” the old table, preventing any new upcoming insertion/extraction operation into/from it to complete. In fact, from now on, any thread invoking a queue operation will be aware that a resize operation is taking place and will start participating in building the new table.

However, we still need to take care of those procedures that have been invoked before the new table has been published. To this goal, once the reference to the new table is public, the procedure invokes both the Block and BlockNext APIs on each PBQ of the old set table (lines R5-R7). Block

Algorithm 3 Reshuffle

```
procedure Reshuffle()
    h ← table
    S′ ← new SetTable(newSize(h))
    CAS(&h->new_table, NULL, S′)
    for pb ∈ h do
        pb.Block()
        e ← pb.BlockNext(pb.head)
    BW ← ComputeNewBW(h)
    h′ ← h->new_table
    CAS(&h′, −1, BW)
    t′ ← h′->table
    l′ ← h′->length
    bw′ ← h′->bw
    for pb ∈ h do
        while True do
            e ← pb.BlockNext(pb.head)
            if e = ⊥ then
                break
            pb.BlockNext(e)
            b′ ← ⌊e.t/bw′⌋
            pb′ ← t′ [b′  mod l′]
            repeat
                old ← h′.current
                if b′ > old.value then
                    break
            until CAS(&h′.current, old, ⟨b′, old.epoch+1⟩)
            if pb.Migrate(e, pb′) then
                Fetch&Add(&h->new_table->size, 1)
    CAS(&table, h, &h->new_table)

procedure checkResize(SetTable h)
    return newSize(h) ≠ 0

procedure newSize(SetTable h)
    if h.size< h.length/2 then
        return min(h.length/2, 1)
    else if h.size> 2h.length then
        return 2h.length
    return 0
```

```
procedure block()
    old_next ← NULL
    while isMarked(old_next, VAL) do
        old ← head.next
        CAS(&pb.next, old, mark(old, MOV))

procedure blockNext(Node n)
    while ⊤ do
        ⟨l, l_next, r⟩ ← Search(n.ts, DEL ∨ INV)
        if unmark(l.next) ≠ r then
            CAS(&l.next, l_next, mark(r, MOV))
        if r = tail then
            return r
        r_next ← r.next
        if isMarked(r_next, VAL) then
            return r
        if CAS(&r.next, mark(r_next,VAL),
                                            mark(r, MOV)) then
            return r

procedure Migrate(Node n, PBQ b)
    c ← Clone(n)
    res ← ⊥
    while n.replica ≠ NULL do
        ⟨l, l_next, r⟩ ← b.Search(n.ts, DEL)
        if l.ts ≠ c.ts then
            c.next ← mark(r, INV)
            if CAS(l.next, unmark(l_next), c)  then
                break
        else
            Free(c)
            c ← l
            break
    res ← CAS(&n.replica, NULL, c)
    tovalidate ← n.replica
    while ⊤ do
        next ← tovalidate.next
        if ¬ isMarked(next, INV) then
            break
        CAS(&tovalidate.next, next, unmark(next))
    CAS(&n.next, mark(next,MOV), mark(next,DEL))
    return res
```

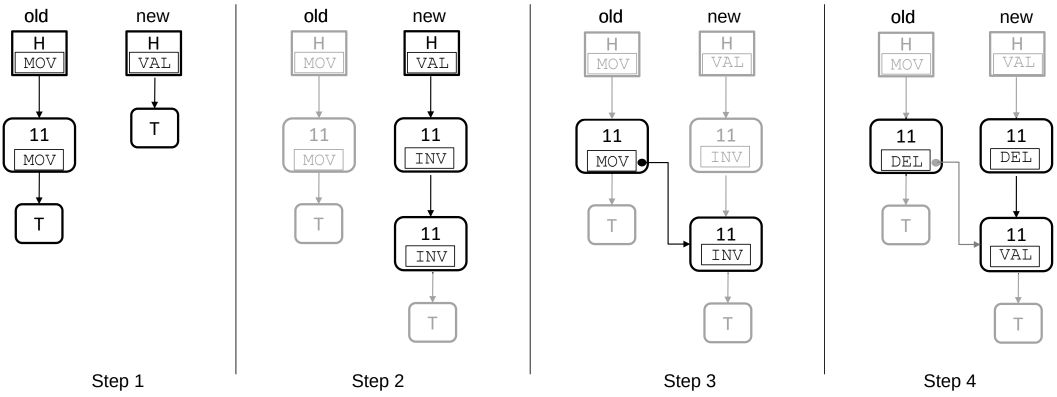


Fig. 4. Main steps of the migration protocol.

makes the insertions of new minimal keys in each bucket fail by returning ABORT. BlockNext makes new extractions fail. In both cases, the failed invocations make the calling threads restart

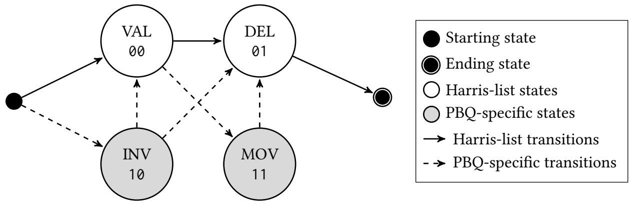


Fig. 5. State machine of nodes in a PBQ.

from scratch and eventually collaborate in the reshuffle phase. As shown in Algorithm 3, these goals are achieved by marking as MOV the head node of the PBQ (this is done by Block) and the first valid item (this is done by BlockNext). As hinted before, this avoids the extraction of the keys of nodes marked as MOV and the insertion of a new item adjacent to a marked one. In fact, the relative CAS are fated to fail since the old value is always a reference marked as VAL instead of MOV (see lines C10 of Algorithm 1 and G7-G12 of Algorithm 2). We note that it is still possible that a prolonged dequeue operation might extract an item that was a minimum in a previous time frame, and it is currently far away from the actual minimum.

At this point, the algorithm computes the new bucket width (line R8) as in the original calendar queue, namely by sampling events and computing their mean separation time, and publishes it with a CAS (line R10). Once the new BW has been decided, the algorithm migrates events from the old set table to the new one. This is done iteratively for each PBQ by marking the highest priority event e and its successor e' (lines R16-R19) as migrating, guaranteeing that both items cannot be extracted due to a dequeue invocation and no insertion of a timestamp k such that $k \in [e.t, e'.t]$ can complete without participating to the reshuffle phase. Before migrating the item e, we update the current variable of the new set table if e is a new minimum (see loop at line R22). Finally, the item e can be migrated from the old PBQ to the corresponding one in the new set table by invoking the relative API of PBQs (line R27). When the PBQ is empty and blocked (line R17), the algorithm proceeds to the next bucket until all items have been migrated. At this point, the old set table is empty and no one can succeed in inserting new items, thus we can exchange it with the new table with a single CAS (line R29).

The node migration is implemented in the procedure Migrate of Algorithm 3 and its main steps are depicted in Figure 4. It adopts a copy-and-validate strategy where items to be migrated into a new PBQ are inserted in possibly multiple invalid copies, one of which will be later validated, and the others will be removed. To this goal, the state INV plays a key role. In particular, after a node has been marked as MOV (step 1 of Figure 4), a thread inserts into the new PBQ an INV copy of the original node (step 2 in Figure 4). As hinted before, this avoids that such a copy can be used in an extraction for a Dequeue procedure. Clearly, since we are non-blocking, multiple threads might create an INV copy of the original node (as shown in step 2 of Figure 4). Consequently, we need to agree on which replica should be considered the valid one and which is just redundant and can be removed. To this goal, we make the original copy point to precisely one replica by setting a pointer via CAS as in step 3 of Figure 4. This kind of publishing allows only one thread to choose its replica. Now, we can validate the unique “master” replica (marking it as VAL), while all the others (including the original copy) can be deleted (marked as DEL like in step 4 of Figure 4).

At this point, we have explored all the allowed state transitions for nodes in PBQs, which are summarized in Figure 5.

3.5 Performance Optimizations

As discussed in Section 1, priority queues are challenging due to their semantics that make the most write-accessed region of the data structure be highly shared among threads and hence CPU cores. Consequently, a large amount of cache-coherency transactions occur during run time, hampering performance. It follows that trying to reduce the impact of these communications has a key role in providing a conflict-resilient data structure. To this end, we introduce one principal optimization, i.e., completely avoiding disconnecting `DEL` items from the buckets used for extractions. In fact, the latter will indeed create cache-coherency traffic since the updated pointer is highly shared among processors.

Further performance optimization is given by reducing the frequency of updates on the `current` variable which is highly accessed in both read and write mode. In particular, this field is read at each operation and written either in case of insertion in a previous bucket or because the current bucket is empty and we need to check the subsequent one. The second case is likely more frequent compared to the former and consequently, reducing its impact is crucial for providing conflict resiliency. To achieve this goal, we need to control the number of events that belong to one virtual bucket on average (denoted as events per bucket or, shortly, as EPB). For instance, the larger EPB is, the more extractions can complete without updating `current`. This can be achieved by simply scaling the bucket width computed by the original sampling algorithm.

Clearly, such an approach has also its drawbacks. In fact, the number of items traversed by a DEQUEUE to find a candidate for extraction increases as the EPB increases. It means that reducing the impact of RMW instructions on caches increases the execution time of an individual extraction. Consequently, we need to face the trade-off between the latency of a dequeue and the overall throughput. This aspect will be deeply discussed in Section 6.

4 LOCK FREEDOM

In order to prove that our algorithm is lock-free, we analyze each method of our proposal, showing that either each thread executing it eventually terminates or it will loop indefinitely due to the progress of other threads.

Both enqueue and dequeue operations invoke internal procedures (e.g., reshuffle), which in turn must guarantee lock freedom to make external procedures (enqueue and dequeue) lock-free. Since it is not trivial to show that the composition of lock-free methods is still lock-free, we prove that the reshuffle algorithm provides a stronger progress condition, i.e., wait-freedom, which ensures that each method invocation completes in a finite number of steps.

Proof structure. We start by showing that the operations on PBQ are lock-free or wait-free. Then, we will prove that the reshuffle algorithm guarantees wait freedom. Finally, since PBQ methods and the reshuffle routine are (at least) lock-free, we can easily show that enqueue and dequeue procedures are lock-free.

LEMMA 1. *Any execution containing invocations to* CONNECT, GETMIN, BLOCK *and* BLOCKNEXT *is lock-free.*

PROOF. CONNECT does not contain any retry loop. Hence, it is a wait-free method as long as the number of items within a bucket is finite—recall that wait freedom implies lock freedom. Under the same conditions, the GETMIN procedure is also wait-free. In fact, it consists of a single loop that scans the PBQ list to find an item to be extracted. However, threads might be stuck in a SEARCH invocation, which performs a traversal of the list, because of continuous concurrent insertions of new items ahead of the scan, which therefore make progress. Consequently, if we consider executions that only include invocations to CONNECT and GETMIN, they are lock-free methods.

Let us now consider the Block method. It contains a unique loop that tries to mark the head as MOV via CAS. This loop terminates if and only if either the CAS succeeds or some other thread has marked the head node as MOV (recall that no transition goes to the INV state, and head nodes are never marked as DEL). Consequently, if the Block invocation is stuck into the retry loop, this is due to an unbounded number of completed insertions at the head of the bucket, namely completed invocations to the Connect procedure, which therefore make progress. On the other hand, the progress of GetMin and Connect is not hampered by Block invocations. In fact, both GetMin and Connect complete by returning ABORT if they find an item marked as MOV by a concurrent and conflicting Block invocation. Consequently, if we consider executions only containing invocations to GetMin, Connect and Block, they are lock-free methods.

Let us now consider BlockNext, which takes as parameter a node *n* marked as MOV and completes when it successfully sets the subsequent VAL node *s* as MOV, or *s* is a tail or *s* is already marked as MOV. If the unique loop of BlockNext does not complete, it means that the CAS for setting *s* as MOV always fails due to a concurrent Connect, GetMin or BlockNext invocation. In any case, some method has completed, hence making progress. Similarly to the Block method, also BlockNext does not hamper the progress of Connect and GetMin. In fact, when such methods get an item marked as MOV, their invocation completes. It follows that considering the executions that contain only invocations to GetMin, Connect, Block and BlockNext, they are lock-free methods. □

Now, we will show that the reshuffle procedure is wait-free. This allows simplifying the lock-freedom proofs of both the enqueue and dequeue procedures by considering such a complex algorithm as a black box that cannot hamper the progress of any calling procedure since it is guaranteed to complete in a finite number of steps.

Lemma 2. *The Reshuffle procedure is wait free.*

Proof. The first step of the reshuffle consists in publishing a new set table with a single-attempt CAS (line R3 of Algorithm 3). This makes every new upcoming dequeue or enqueue operation participate in the reshuffle phase until each item is migrated from the old to the new set table, and the two are swapped. When such a publishing step has completed, the number of threads that are currently executing a dequeue or an enqueue and that will not participate in the reshuffle algorithm—since their operation is not upcoming with respect to the CAS—is finite. Let T be the number of these pending calls.

After publishing the new set table, the reshuffle protocol starts by blocking each PBQ in the old set table and each event with the minimum timestamp in each bucket. These two operations are performed by invoking the Block and BlockNext methods, which are lock-free. Since all threads that execute a reshuffle procedure will invoke Block and BlockNext on each bucket, both these methods might be blocked only due to concurrent Connect and GetMin invocations as discussed in Lemma 1. Those Connect and GetMin invocations can be performed only by those T pending invocations of enqueue and dequeue operations, which are not participating in the reshuffle protocol. Eventually, these T threads will complete their respective calls of the Connect and GetMin methods by either successfully inserting/extracting an item or because they access an item marked as MOV. This ensures that each thread executing a reshuffle will end the first cycle (lines R5-R7). It follows that there is a moment from which it is guaranteed that the number of enqueued items in the old set table, which is reshuffled, does not increase, i.e., it is bounded by a constant.

At this point, every thread tries to migrate each item from the old set table to the new one. Before such a step, a thread updates the current variable of the new set table with a value corresponding to the item to be migrated. Since each thread tries to migrate each item, the number of items is bounded by a constant. Such a loop is guaranteed to complete in a finite number of steps if and only if the migration is wait-free. Also, at the end of each migration, an item has been removed from the old set table and added to the new one. It follows that each PBQ is guaranteed to eventually be emptied if and only if the migration is wait-free. Under this assumption, each thread completes the loop at line R15, showing that the reshuffle procedure provides wait-freedom. Thus, to conclude our proof, we need to show that the migration of a single item is wait-free.

Migration consists of: i) creating a copy of an item (line M2); ii) inserting such a copy marked as `INV` in the new set table (lines M4-M13); iii) making the original item point to the just inserted invalid copy (line M14); iv) marking as `DEL` and as `VAL` the primary and replica nodes, respectively (lines M16-M20). Steps i) and iii) do not contain any loop, so they terminate eventually. Conversely, step ii) and iv) are performed within a retry loop.

Since the new table is not used to add/remove items until the reshuffle is completed, a thread retries step ii) or step iv) if and only if concurrent reshuffle invocations conflict with each other while updating a `next` field of a common item during either step ii) or step iv), namely while inserting their own copy or validating a replica—marking it as `VAL`. Since such updates are performed via `CAS`, only successful ones can cause other threads to fail and retry.

Let us consider the case where a thread is stuck indefinitely due to other threads performing step iv) successfully. On the one hand, a thread that successfully validates an item i, makes all other threads trying to validate i complete. On the other hand, the number of successful executions of step iv) is bounded by the number of nodes that can be validated, i.e., those referred to by the original copy. Clearly, such a number of items is the size of the old set table, which has been shown to be bounded by a constant. Consequently, the number of retries that a thread performs in step ii) or iv) is bounded by a constant because other conflicting threads successfully perform step iv).

Let us consider the case in which a thread is stuck due to other threads performing step ii) successfully. To make this happen, we need an unbounded number of copies to be inserted into the new set table. This operation is performed via `CAS` (line M8). Eventually, at least one of these insertions will proceed with step iii) (line M14), avoiding that other threads create additional copies. Additionally, if a thread finds an `INV` copy of the same item it wants to migrate, it releases its own copy and tries to validate the one found during the scan (see lines M10-M13). This guarantees that, for a given item i, the number of successful replica insertions is finite. Since the number of items is bounded by a finite constant, the number of successful executions of step ii) is also bounded by a constant. Consequently, the number of retries that a thread performs in step ii) or iv) is bounded by a constant due to other conflicting threads successfully performing step ii).

Since step i) and iii) do not contain any loop, and step ii) and iv) can be retried a finite number of times, each invocation of the reshuffle procedure is guaranteed to complete in a finite number of steps eventually. Thus, the reshuffle algorithm is wait-free. □

Now, we can show that queue procedures are lock-free.

Lemma 3. *The enqueue procedure is lock-free.*

Proof. The enqueue has two cycles: a) the one executed to update `current` (lines E15-E20 of Algorithm 1) and b) the one which includes the invocation to Connect, Reshuffle and checkResize (lines E4-E7).

We start by analyzing loop b). The checkResize procedure is trivially wait-free since it does not contain any loop. Since also the Reshuffle procedure is wait-free by Lemma 2, cycle b) might loop unboundedly only in the following scenarios: S1) each checkResize invocation detects that a resize is needed and S2) Connect always returns `ABORT`.

In S1), some other thread is completing its Enqueue or Dequeue invocation and, thus, the procedure is not making progress due to the progress of other threads.

In S2), some other thread has updated a field that causes the Connect procedure to always abort. Since reshuffles are wait-free, if such updates are performed by some reshuffle phase, it follows that the data structure is continuously reshuffled because an unbalanced number of enqueues or dequeues has completed. If the Connect invocation fails due to a concurrent GetMin the latter has successfully marked a node as `DEL`. When this happens, the calling Dequeue will complete. Consequently, the Enqueue procedure does not complete because of the completion of some other Dequeue. Finally, a Connect invocation might fail due to successful concurrent Connect calls. However, we cannot conclude that the respective Enqueue has completed, but only that this thread has made a step forward in its computation by passing from loop b) to loop a). In fact, such an Enqueue procedure might be stuck in loop a).

Consequently, we can conclude our proof by showing that a thread might be stuck in loop a) only because other Enqueue or Dequeue invocations have completed.

A thread A iterates unboundedly in this loop if and only if the `CAS` at line E20 always fails, and the condition at line E20 is always false. It follows that each read operation of `current` returns an index greater than the one corresponding to the VB updated by A. This scenario is feasible due to either concurrent enqueues that modify `current` and then complete (making progress) or by concurrent dequeues that increase the current index (line D18 of Algorithm 2). If those dequeues continue to increase `current` without making progress, it means that every VB is empty, a condition that eventually triggers a reshuffle. When this happens, thread A will eventually execute the retry loop for updating `current` of the old set table in isolation and completes eventually. □

Lemma 4. *The dequeue procedure is lock-free.*

Proof. The dequeue has one main cycle that includes invocations to Reshuffle and GetMin functions. The Reshuffle call is guaranteed to complete as it is a wait-free method by Lemma 2. The GetMin procedure does not return only because an unbounded number of successful insertions prevent the scan from completing (see Lemma 1). Consequently, we need to consider only the conditions that make a thread continuously loop in the main cycle. In particular, a thread A iterates unboundedly if and only if `current` points to buckets such that the subsequent invocations of GetMin by A fail a) because they are empty (returning $\perp$ or $\emptyset$); b) due to concurrent insertions (returning `ABORT`); c) due to continuous reshuffles.

Case b) implies that some enqueue has completed, hence making progress, and thus it is not relevant for the discussion (the thread is starving because some other thread is making progress).

Case c) implies that both enqueue and dequeue calls are completing and triggering reshuffles, thus it is not relevant for the discussion.

Case a) occurs if and only if either i) other threads extract items faster than A or ii) A always finds empty PBQs because the queue is actually empty, but the calendar size is greater than one. In case i), someone is making progress, thus it is not relevant for the discussion. Conversely, case ii) could lead to a blocking scenario in which thread A waits for a new insertion to complete. This is impossible, because, if such enqueue is never issued, thread A repeatedly triggers reshuffles (line D20 of Algorithm 2) until it meets the empty condition and completes (line D15). □

Theorem 1. *The LFCQ data structure is lock-free.*

Proof. Since both enqueue and dequeue procedures are lock-free, by Lemmas 3 and 4 respectively, the LFCQ data structure is lock-free. □

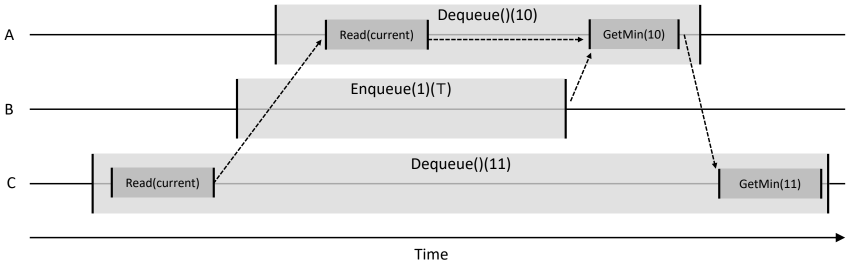


Fig. 6. History that cannot be linearized with fixed linearization points.

5 LINEARIZABILITY

A classic approach for proving the safety of a non-blocking data structure consists in individuating *linearization points* of its methods into specific atomic instructions. Thus, the algorithm designer only needs to prove that the ordering of concurrent operations obtained by using those linearization points as representative of the materialization of the operations is compatible with the semantics of the managed data structure.

The LFCQ methods have a finite number of instructions that work on shared data, and all of them are atomic (loads or CAS), but we cannot use them as fixed linearization points. As an example, let us consider an initialization of the queue with 2 keys, 10 and 11, and the timeline in Figure 6. For the sake of simplicity, we collapse invocation and termination of a procedure in a single signature. In particular, $D()(v)$ (or $D(v)$) denotes a Dequeue returning the value v and $E(e)(w)$ is an Enqueue of the event e that returns w, denoted with $E(e)$ when $w = \top$. We have two successful dequeues extracting items (denoted as $D()(10)$ and $D()(11)$ respectively) and one enqueue inserting the key value 1. The three methods are concurrent to each other, and we can easily find an equivalent sequential history [19] which is linearizable, like $D()(10) \prec D()(11) \prec E(1)(\top)$. However, by the algorithm structure, dequeues have the successful CAS during the invocation of the GetMin procedure and have the preceding atomic read of `current`. Choosing CAS as the linearization point will result in the following sequential history $E(1)(\top) \prec D()(10) \prec D()(11)$ which is not legal since it violates the priority queue semantics. On the other hand, choosing the read of `current` as the linearization point for dequeues will result in $D()(11) \prec D()(10)$ which is not legal regardless of the enqueue positioning.

To prove that our LFCQ algorithm is correct, we will follow an aspect-oriented approach [5, 11]. It consists of proving the algorithm's specific invariants to show that each possible run it generates is correct. We are interested in proving that our algorithm implements the priority-queue semantics, which is a combination of set semantics with an additional constraint on the values obtained by extractions. The authors in [11] have shown that the problem of correctly implementing the set semantics can be reduced to ensuring that the following violations cannot occur:

MultiExt: elements dequeued multiple times;
MultiDeq: a dequeue extracts multiple items;
NotEnq: dequeued elements without a corresponding enqueue;
FalseEm: a dequeue returns $\perp$ even though the set was never logically empty during its execution.

Consequently, avoiding the above violations while jointly guaranteeing that the extracted element has the highest priority leads to the correctness of the priority queue algorithm. The additional violation to be avoided can be stated as follows:

NotOrd: a dequeue returns a value even though it was never the minimal key during its execution.

Proof structure. The proof follows an incremental approach. We prove the linearizability of LFCQ in Section 5.1. In particular, Section 5.1.1 shows that LFCQ does not have **MultiDeq**, **MultiExt** and **NotEnq** violations. The absence of **NotOrd** and **FalseEm** in the LFCQ algorithm are discussed in Sections 5.1.2 and 5.1.3 respectively. Finally, we conclude the proof in Section 5.1.4.

5.1 Proving the Priority Queue Semantics of LFCQ

5.1.1 *Avoidance of* **MultiDeq**, **MultiExt** *and* **NotEnq** *violations.*

LEMMA 5. *An item can be stored in up to one* VAL *node at a time.*

PROOF. Items are first inserted as VAL in a PBQ by invoking the CONNECT API during an ENQUEUE. Inspecting the code of Algorithm 1, we have a single successful CONNECT per each completed ENQUEUE, which connects a single VAL node in a PBQ. Finally, items can be migrated from one PBQ to another one. To this end, the node must be first marked as MOV, a transition that can be applied once per node. Then, the MIGRATE API can validate only one replica (the one reachable from the dedicated field within the original node) by applying the INV to VAL transition. □

LEMMA 6. **MultiDeq** *violations cannot occur in any execution obtained by running the LFCQ algorithm.*

PROOF. A dequeue operation extracts an item when the corresponding GETMIN extracts an item from an individual PBQ. To reach this goal, the GETMIN needs to apply the transition VAL to DEL on the target node via CAS, which is executed once per dequeue. □

LEMMA 7. **MultiExt** *violations cannot occur in any execution obtained by running the LFCQ algorithm.*

PROOF. By Lemma 5, we know that an item can be stored in up to one VAL node at a time. It follows that the VAL to DEL transition can be applied at most once for a given enqueued item. □

LEMMA 8. **NotEnq** *violations cannot occur in any execution obtained by running the LFCQ algorithm.*

PROOF. A DEQUEUE returns an item that has been extracted by the last successful invocation of GETMIN. By construction, nodes are inserted into a PBQ only by CONNECT and by MIGRATE procedures. From Lemma 5, we know that a migrated item was previously inserted by a previous CONNECT. Since CONNECT procedures are invoked only by ENQUEUE methods, the **NotEnq** violation cannot occur. □

5.1.2 *Avoidance of the* **NotOrd** *violation.*

We now prove that the **NotOrd** violation cannot occur. We follow an incremental approach, where we first show that **NotOrd** does not occur when no reshuffle occurs. Then, we prove that nothing wrong happens during the reshuffle phases.

Preliminaries. The authors in [3] have shown that there is a complete characterization of the **NotOrd** violation. In particular, given a history compliant with the set semantics, there is an algorithm to check if **NotOrd** occurs (the pseudocode is shown in Algorithm 4). The method iteratively removes invocation/completion pairs from the original history until the resulting history is empty, meaning that the original history is correct. The first step consists in removing pairs related to the failed extractions, namely those that have detected the queue as empty. Then, it removes pairs related to the enqueue operations that have inserted the lowest priority element (the maximum timestamp appearing in the history), but there is no corresponding extraction. Then, since the set semantics guarantees that failed DEQUEUE and ENQUEUE operations do not affect the queue state, they can be removed from the original history (they are correct by hypothesis). At this

Algorithm 4 Procedure to check **NotOrd** presence

```
procedure checkHistory($H$)
    $H' \leftarrow H$
    repeat
        Remove from $H'$ any $E(v)(\bot)$
        Remove from $H'$ any $D()(\bot)$
        Remove from $H'$ any $E(v)$ such that $v$ is a maximal priority in $H' \wedge \nexists D(v) \in H'$
        Get current maximal priority $x$ in $H'$
        $G \leftarrow$ buildLeftRightConstraintGraph($x, H'$)
        $res \leftarrow$ hasCycle($G$)
        if $\neg res$ then
            return $res$
        $H' \leftarrow H'/\{E(x), D(x)\}$
    until $H' = \emptyset$
    return $res$
```

point, it considers the lowest priority x in the current history and evaluates if the relative extraction is correct. If the extraction is not correct, the algorithm concludes that the original history is not correct. Otherwise, it is correct, and the algorithm removes both the insertion and the extraction of x and restarts targeting the new lower priority.

In order to check if the extraction of the lowest priority x is correct, the authors reduce the problem into looking for a cycle in a graph built from the history. The graph, called *left-right-constraint graph of* x (denoted as $LRCG(x)$), is built in a manner such that if a cycle is present, it follows that x has never been a minimum during the execution of $D()(x)$, thus preventing linearizability. In other words, it represents the relationship of $D()(x)$ with all other operations that prevent x from being the highest priority during the execution of $D()(x)$ and make the latter not linearizable. Each node in the graph represents a procedure in H, while edges represent a relation between two operations. We can distinguish three kinds of edges that might appear in an LRCG[1]:

pr: $A \xrightarrow{pr} B$ means that $A \prec_H B$, namely A precedes B in the real-time order of the analyzed history H;

val: $D()(a) \xrightarrow{val} E(b)(\top)$ means that $D()(a)$ has extracted the item inserted by $E(b)(\top)$ and thus $a = b$;

lpr: $A \xrightarrow{lpr} B$ means that A should logically precede B according to the semantics of the priority queue (for example $E(x) \xrightarrow{lpr} D(x)$).

The left-right-constraint graph of x can be built by adopting the following rules:

(1) initialize the graph to $E(x)(\top) \xrightarrow{lpr} D(x)$;
(2) if $E(v) \prec_H E(x)$ then add the following edges $D(v) \xrightarrow{val} E(v) \xrightarrow{pr} E(x)$;
(3) if $E(v) \prec_H D(x)$ then add the following edges $D(v) \xrightarrow{val} E(v) \xrightarrow{pr} D(x)$;
(4) if $E(v_2) \prec D(v_1) \wedge D(v_1) \in G$ then add the following edges $D(v_2) \xrightarrow{val} E(v_2) \xrightarrow{pr} D(v_1)$;
(5) if $D(x) \prec D(v)$ then add the edge $D(x) \xrightarrow{pr} D(v)$;
(6) if it exists an enqueued item with priority v which is never extracted add the following edge $D(x) \xrightarrow{lpr} D(v)$.

Once we have built the graph with the rules mentioned above, we have to look for a cycle. If it exists, it means that the extraction of x is not legal, and thus the history is not linearizable otherwise we can remove both $E(x)(\top)$ and $D()(x)$ and repeat the check for the next lowest priority.

[1] The original specification of LRCG does not resort to different kinds of edges. These have been introduced in our work to simplify the exposition of our proofs.

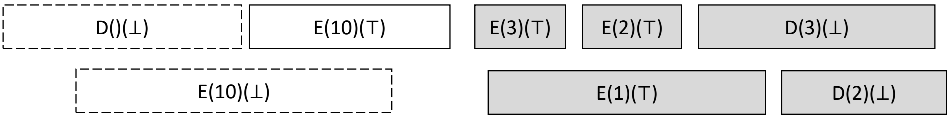


Fig. 7. Timeline of a history with **NotOrd** violation.

To make this concept clear, consider the timeline in Figure 7 describing an execution running on a priority queue, which implements set semantics correctly. Dashed blocks correspond to procedures that do not affect the internal state of the priority queue, and hence they can be ignored for checking the correctness of successful extractions. According to line CH6 of Algorithm 4, the successful Enqueue of 10 can be removed since the item has the lowest priority in the history, and no extraction of it exists in the history. In other words, we only have to consider grey routines for checking correctness. In particular, we need to check the correctness of each extraction, starting from the lowest priority, which is 3 in our case. According to rule 1, we initialize the graph as follows:

$$D(3) \xleftarrow{\text{lpr}} E(3)$$

According to rule 3, we can add $E(2)$ and $D(2)$ nodes and the relative outcoming edges

$$D(2) \xrightarrow{\text{val}} E(2) \xrightarrow{\text{pr}} D(3) \xleftarrow{\text{lpr}} E(3)$$

Then, rule 4 allows us to add nodes and edges of procedures related to the value 1

$$D(1) \xrightarrow{\text{val}} E(1) \xrightarrow{\text{pr}} D(2) \xrightarrow{\text{val}} E(2) \xrightarrow{\text{pr}} D(3) \xleftarrow{\text{lpr}} E(3)$$

Since there is no extraction of 1, we connect $D(1)$ to $D(3)$ according to rule 6

$$\begin{tikzcd}D(1) \arrow[r, "\text{val}"] & E(1) \arrow[r, "\text{pr}"] & D(2) \arrow[r, "\text{val}"] & E(2) \arrow[r, "\text{pr}"] & D(3) \arrow[llll, bend left, "\text{lpr}"] & E(3) \arrow[l, "\text{lpr}"']\end{tikzcd}$$

Now, we cannot apply any other rule and the LRCG(3) is completed. Since it contains a cycle, the algorithm concludes that the extraction of 3 violates the priority queue semantics and the history is not legal, namely there is no equivalent sequential history that is legal and preserves the real-time order of the original history.

The proof structure. In order to prove the absence of the **NotOrd** violation in a history of LFCQ not including reshuffle phases, we will show that starting from LRCG(x) we can build a *precedence graph of* x (denoted $PG(x)$) such that it preserves the cycles of LRCG(x) (if there is any), its edges only belong to the real-time precedence relation, and its nodes are atomic operations. Then, we proceed by contradiction. In fact, assuming the presence of a **NotOrd** violation requires a cycle in LRCG(x), which implies a cycle in the precedence graph $PG(x)$. However, the atomic instructions in the precedence graph are totally ordered, which implies the impossibility of the cycle occurrence. Finally, to conclude the proof, we will show that nothing bad happens during a reshuffle phase.

In order to build a precedence graph from an LRCG, we need some kind of operator to replace edges and nodes from LRCG with edges belonging to the precedence relation and nodes corresponding to atomic instructions.

As the first step, Lemma 9 shows that there is a logical implication that correlates the extraction of a priority x and the insertion of a priority k when $k < x$ and they belong to the same bucket, namely $b_x = b_k = b$, where b_i is the VB of priority i. This allows us to replace specific nodes in the LRCG with the atomic instructions that connect and extract an item within the same PBQ and use edges from the precedence relationship.

Then, Lemma 11 proves the same property for the case of two priorities belonging to two different buckets. This relies on a specific invariant guaranteed by reads of the current variable, to be shown in the preliminary Lemma 10.

Lemmas 9 and 11 provide tools to manipulate an LRCG, which will be exploited in Lemma 12 to show that if a **NotOrd** violation occurs, the atomic instructions provided by the hardware are not totally ordered, i.e., they are non-atomic, thus showing that **NotOrd** cannot occur in our algorithm when no reshuffle occurs.

At this point, after proving that reshuffle phases do not overlap with each other in Lemma 13, we prove in Theorem 2 that **NotOrd** violations cannot occur even in the presence of reshuffle phases.

The proof. In our proofs, we will use the following notations: reads of current with the relative epoch are denoted as $R(index, epoch)$; $F(index, epoch)$ is the atomic update of current made by enqueues (line E20 of Algorithm 1); $I(index, epoch)$ is the atomic update of current made by dequeues (line D18 of Algorithm 2); C and G are abbreviation for the Connect and GetMin methods of the PBQ API. When successful, such procedures will be considered as materialized at the atomic CAS contained in their execution which inserts/extracts the relative item.

Lemma 9. *Assume that no reshuffle occurs in the history. Let* x *and* k *be two priorities such that* $\mathrm{k} < \mathrm{x}$ *and* $\mathrm{b_x} = \mathrm{b_k} = \mathrm{b}$*. If the insertion of* k *precedes the insertion of* x *or the extraction of* x*, then the successful* CAS *of the* GetMin *extracting* k *precedes the one extracting* x.

Proof. By inspecting the pseudo-code of enqueue and dequeue procedures (Algorithms 1 and 2), we know that there must be a unique successful invocation of a PBQ procedure for each successful API invocation of LFCQ. In more detail, $E(k)(\top)$, $E(x)(\top)$ and $D()(x)$ executions contain the execution of one successful PBQ API, namely $C(b, k, t_k)(\top)$, $C(b, x, t_x)(\top)$ and $G(b, t_D^x)(x)$ respectively—recall that the last parameter is the epoch used for the PBQ method invocation. Consequently, we know that

a) $E(k) \prec D()(x) \longrightarrow C(b, k, t_k)(\top) \prec G(b, t_D^x)(x)$

or

b) $E(k) \prec E(x) \longrightarrow C(b, k, t_k)(\top) \prec C(b, x, t_x)(\top)$.

For case a), we know that the extraction A of x has traversed all the items preceding x in the VB. Since the Connect procedure has completed before the GetMin invocation, it follows that the item k belonged to the VB considered by A. If the latter has extracted x, it means that k was either marked as DEL or disconnected from the list. Consequently, k has been extracted before the extraction of x has been completed, proving the lemma.

For case b), we consider all the possible interleavings of the three procedures $E(k), E(x)$ and $D()(x)$. If $E(x) \prec D()(x)$ in the real time order, then $E(k)$ precedes $D()(x)$. Hence, we fall in case a) which has already been shown to prove the lemma. If $D()(x) \prec E(k)$, it means that a dequeue has extracted a non-enqueued item, a scenario which is ruled out by Lemma 8. Hence, we need to focus only on executions such that the Dequeue is concurrent with the Enqueue of both k and x. In particular, the successful GetMin invoked by the considered Dequeue must be concurrent to both the Connect invocations that inserted k and x. This has several implications on the parameter passed to the successful PBQ method invocations. In more detail, we have that:

- $t_x \leq t_D^x$ otherwise the GetMin could not extract x;
- $t_D^x \leq t_x$ since GetMin got a pair index/epoch from a read of current occurred before the one made by the insertion of x.

It follows that $t_x = t_D^x$, implying that no epoch update has been performed between the read of current, which has returned a value indicating b as the current bucket for extraction, and the read of `current` preceding the Connect inserting x. It means that the Enqueue of k has completed before both the above-mentioned reads, otherwise the epoch should have been updated by either the Enqueue of k (it has inserted in the current bucket b) or by another Enqueue that has inserted in a previous bucket $b' < b$, which is impossible. In other words, the Enqueue of k terminates before the successful attempt of GetMin by $D()(x)$. Consequently, all feasible scenarios of case b) can be reconducted to case a) proving the lemma. □

From the previous lemma, we can easily prove what follows:

Corollary 1. *Let x and y be two priorities such that $E(x) \prec E(y)$ and $x < y$. It does not exist any successful invocation of* GetMin *that extracts y, which is concurrent with both the successful* Connect *invocations that inserted x and y.*

Proof. Derives directly from case b) in Lemma 9. □

In order to prove the same statement of Lemma 9 when considering a couple of priorities that belong to different buckets, we need a preliminary lemma. It shows the roles of `current` and epochs in our algorithm as a "glue" among different VBs, allowing extractions not to care about what happens in previous VBs.

Lemma 10. *Assume that no reshuffle occurs in the history. The reading of the value* ⟨b, t⟩ *of current is an indivisible point in time where each item inserted in a previous bucket of* b *has been extracted or its insertion in not completed.*

Proof. By construction, the first read of `current` returns $\langle 0,0\rangle$. Then, each modification of `current` has been applied with a `CAS` either to increase the epoch or the current bucket. We denote these `CAS` instructions as $F(a,b)$ and $I(a,b)$ respectively, where $\langle a,b\rangle$ is the new value assumed by `current`. Consequently, there is a sequence S of updates of `current` the leads from $\langle 0,0\rangle$ to $\langle b,t\rangle$:

$$S = \{[W(\_,\_)]^m .\ W \in \{I,F\}\}\ .\ \ R(0,0) \prec S \prec R(b,t)$$

where $R(x,y)$ is a read of `current` returning $\langle x,y\rangle$. We prove the statement by induction on the number N_F of occurrences of F in S.

Case $N_F = 0$) It follows that S is a sequence of atomic increments that makes `current` point to the next bucket and the epoch be still 0 ($t = 0$)

$$S = \left[\prec_{i=1}^{b} I(i,0) \right]$$

We know that each increment of bucket $I(i,0)$ is preceded by a failed extraction from bucket $i-1$. Consequently:

$$\left[\prec_{i=1}^{b} [G(i-1,0)(\epsilon) \prec I(i,0)] \right] \prec R(b,0)$$

where $\epsilon \in \{\bot, \emptyset\}$—this is because the GetMin API can report the empty VB in two different ways. We know that an extraction $G(i,0)$ returns ϵ if the virtual bucket i was traversed without finding any valid node for extractions. Moreover, the construction of the algorithm implies that `current` must point to a virtual bucket before an extraction can use it as a parameter, namely $I(i,0) \prec G(i,0)(\epsilon)$. It follows that any item e belonging to the VB $i \in [0,b)$ and whose insertion $E(e)(\top) \prec I(i,0) \prec G(i,0)(\epsilon) \prec R(b,0)$, has been extracted when last $G(i,0)(\epsilon)$ completes. On the other hand, any insertion $E(e)$ concurrent to an increment of `current` $I(i,0)$ also requires an increment of epoch to complete, which does not occur until $R(b,0)$ by hypothesis. This implies that $E(e)(\top)$ is still pending when $R(b,0)$ has been evaluated.

Case $N_F = t$) True by inductive hypothesis.

Case $N_F = t + 1$) Let S be a sequence such that:

$$R(0,0) \prec S \prec R(b, t+1)$$

Let S' be a sequence containing all the operations preceding the last F in S. Thus, we can state:

$$S = S' \prec F(h, t+1) \prec \left[\prec_{i=h+1}^{b} I(i, t+1) \right].$$

Moreover, we know that each F has to be preceded by a read to obtain the old value for a CAS. Consequently:

$$S' \prec R(k,t) \prec F(h, t+1) \prec \left[\prec_{i=h+1}^{b} I(i, t+1) \right] \prec R(b, t+1),$$

where $k \geq h$. We also know that each increment of `current` requires its reading for getting the old value, specifically:

$$R(k,t) \prec F(h, t+1) \prec R(h, t+1) \prec \left[\prec_{i=h+1}^{b} I(i, t+1) \right] \prec R(b, t+1).$$

By inductive hypothesis, we know that each insertion in a bucket preceding the k-th one is still pending or completed, and its item has been extracted before $R(k, t)$ occurs. Since $h \leq k$, the same also holds for insertions in a bucket preceding the h-th one. Namely, they are either completed, and their items have been extracted, or they are still pending when $R(k, t)$ occurs.

Consider the insertions in a bucket $j < h < k$ such that they are alive immediately after $R(k, t)$ and suppose that they have completed before $R(h, t + 1)$. They require to update `current` to make it point to the j-th bucket, to increase epoch and to complete. This is impossible otherwise the total number of epochs will be greater than $t + 1$. Consequently, they must be alive when both $F(h, t + 1)$ and $R(h, t + 1)$ occur. It means that the statement holds for $R(h, t + 1)$ as no epoch updates have occurred. We can show that the statement holds for $R(b, t + 1)$ by following the same reasoning for the case $N_F = 0$. □

Thanks to the last lemma, we can prove what follows:

Lemma 11. *Assume that no reshuffle occurs in the history. Let* x *and* k *be two priorities such that* k < x *and* $\mathrm{b}_\mathrm{k} < \mathrm{b}_\mathrm{x}$. *If the insertion of* k *precedes the insertion of* x *or the extraction of* x, *then the successful* CAS *of the* GetMin *extracting* k *precedes the read of* `current` *used to take the parameters for the* GetMin *extracting* x.

Proof. By inspecting the code of Dequeue (see Algorithm 2) we know that the successful invocation of GetMin must be preceded by a read of `current`. Then, Lemma 10 shows that any item belonging to a bucket preceding the one returned by a read R of `current` has been already extracted when R occurs if the Enqueue inserting it has completed before such a read. This concludes our proof when $E(k) \prec D(x)$.

Now, we need to show that the statement holds when $E(k) \prec E(x)$. However, when considering the possible interleavings of Dequeue steps, Corollary 1 restricts the set of admissible executions to those where both the successful GetMin invocation and its previous read of `current` must occur after the completion of $E(k)$. As in the previous case, Lemma 10 concludes the proof. □

Lemma 12. **NotOrd** *violations cannot happen for any history without failed extractions (denoted as $D()(\bot)$) generated by running the algorithm without reshuffle.*

Proof. Suppose by contradiction that the **NotOrd** violation occurs. By hypothesis, we know that there is a cycle in the left-right-constraint graph of a minimal priority x as the one shown in the following graph:

$$\begin{tikzcd}
E(a) \arrow[r, "\text{pr}"] & D(b) \arrow[r, "\text{val}"] & E(b) \arrow[r, "\text{pr}"] & \cdots \arrow[r, "\text{pr}"] & D(k) \arrow[r, "\text{val}"] & E(k) \arrow[r, "\text{pr}"] \arrow[d, "\text{pr}"'] & E(x) \arrow[dl, "\text{lpr}"'] \\
D(a) \arrow[u, "\text{val}"] & & & & & D(x) \arrow[lllll, "\text{lpr} \vee \text{pr}"'] &
\end{tikzcd}$$

This cyclic graph is exhaustive since:

- incoming edges of $E(x)$ nodes are given only by rule 2;
- outcoming edges of $E(x)$ nodes are given only by rule 1;
- incoming edges of $D(x)$ nodes are given only by rules 1 and 3;
- outcoming edges of $D(x)$ nodes are given only by rules 5 and 6;
- the cycle can be built only by concatenating edges added with rule 4.

Consider the edges relating $D(k), E(k), E(x)$ and $D(x)$. We have two cases:

a) $b_k < b_x = b \longrightarrow G(b_k, t_D^k)(k) \prec R(b_x, t_D^x)$ by Lemma 11;
b) $b_k = b_x = b \longrightarrow G(b_k, t_D^k)(k) \prec G(b_x, t_D^x)(x)$ by Lemma 9.

Consequently, we can build two graphs by replacing some edges of the left-right-constraint graph as:

$$\begin{tikzcd}
E(a) \arrow[r, "\text{pr}"] & \cdots \arrow[r, "\text{val}"] & E(j) \arrow[r, "\text{pr}"] & R(b_k, t_D^k) \arrow[r, "\text{pr}"] & G(b_k, k, t_D^k)(k) \arrow[d, "\text{pr}"'] \\
D(a) \arrow[u, "\text{val}"] & & G(b, t_D^x)(x) \arrow[ll, "\text{pr} \vee \text{lpr}"'] & & R(b, t_D^x) \arrow[ll, "\text{pr}"']
\end{tikzcd}$$

$$\begin{tikzcd}
E(a) \arrow[r, "\text{pr}"] & \cdots \arrow[r, "\text{val}"] & E(j) \arrow[r, "\text{pr}"] & R(b, t_D^k) \arrow[r, "\text{pr}"] & G(b, t_D^k)(k) \arrow[dll, "\text{pr}"'] \\
D(a) \arrow[u, "\text{val}"] & & G(b, t_D^x)(x) \arrow[ll, "\text{pr} \vee \text{lpr}"'] & & R(b, t_D^x) \arrow[ll, "\text{pr}"']
\end{tikzcd}$$

Thanks to Lemma 11, we can ignore nodes related to k for case a) (e.g., the above left graph), and hence add an edge of the precedence relation from $E(j)$ to $R(b, t_D^x)$. Since $G(b_k, t_D^k)(k) \prec G(b_x, t_D^x)(x)$ in both cases a) and b), we can consider one general graph structured as the following:

$$\begin{tikzcd}
E(a) \arrow[r, "\text{pr}"] & D(b) \arrow[r, "\text{val}"] & \cdots \arrow[r, "\text{val}"] & E(h) \arrow[r, "\text{pr}"] & D(j) \arrow[r, "\text{val}"] & E(j) \arrow[r, "\text{pr}"] & R(b, \_) \arrow[d, "\text{pr}"'] \\
D(a) \arrow[u, "\text{val}"] & & & & G(b, t_D^x)(x) \arrow[llll, "\text{lpr} \vee \text{pr}"'] & & G(b, t_D^k)(k) \arrow[ll, "\text{pr}"']
\end{tikzcd}$$

Now, let us consider the relationship obtained from the graph above by taking into account nodes corresponding to j, k and x:

$$\begin{tikzcd}
\cdots \arrow[r, "\text{val}"] & E(h) \arrow[r, "\text{pr}"] & D(j) \arrow[r, "\text{val}"] & E(j) \arrow[r, "\text{pr}"] & R(b,\_) \arrow[r, "\text{pr}"] & G(b,t_D^k)(k) \arrow[r, "\text{pr}"] & G(b,t_D^x)(x) \arrow[r, "\text{lpr} \lor \text{pr}"] & \cdots
\end{tikzcd}$$

According to the position of b_j, we can apply different transformations to the above-mentioned portion of the graph. In particular, if $b_j < b$, Lemma 11 tells us that the extraction of j has been completed when the read $R(b,\_)$ occurs, allowing us to apply the following transformation that completely shortcuts nodes regarding j:

$$\begin{tikzcd}
\cdots \arrow[r, "\text{val}"] & E(h) \arrow[r, "\text{pr}"] \arrow[rrr, bend right, "\text{pr}"'] & R(b_j,t_D^j) \arrow[r, "\text{pr}"] & G(b_j,t_D^j)(j) \arrow[r, "\text{pr}"] & R(b,\_) \arrow[r, "\text{pr}"] & G(b,t_D^k)(k) \arrow[r, "\text{pr}"] & G(b,t_D^x)(x) \arrow[r, "\text{lpr} \lor \text{pr}"] & \cdots
\end{tikzcd}$$

Conversely, when $b_j = b$, the scenario becomes slightly more complex since we need to consider the specific values of j and k. In particular, if $j \leq k$, we can obtain the following by exploiting Lemma 9:

$$\begin{tikzcd}
\cdots \arrow[r, "\text{val}"] & E(h) \arrow[r, "\text{pr}"] & R(b,t_D^j) \arrow[r, "\text{pr}"] & G(b_j,t_D^j)(j) \arrow[rr, bend left, "\text{pr}"] \arrow[rrr, bend right, "\text{pr}"'] & R(b,\_) \arrow[r, "\text{pr}"] & G(b,t_D^k)(k) \arrow[r, "\text{pr}"] & G(b,t_D^x)(x) \arrow[r, "\text{lpr} \lor \text{pr}"] & \cdots
\end{tikzcd}$$

and, also in this case, we can shortcut some nodes, in particular those corresponding to k. The last case where $k \leq j \leq x$ requires additional considerations. In fact, Lemma 9 does not tell us anything about insertions and extractions, of which we do not know the relative precedence order. However, it tells us that the extraction of x has seen the effect of the extraction of k. Moreover, from the precedence graph we know that the insertion of j is completed before extracting k. Consequently, since PBQ implements set semantics and is an ordered list, we know that the scan performed during the extraction of x has traversed the nodes looking for a key belonging to b. Since it has seen the effect of the CAS extracting k, it should have seen the effect of the previous CAS inserting j. In such a case, it should have extracted j, but since it has extracted x and we know $j < x$, we know that j was marked as DEL or disconnected. In other words, the extraction of x has seen the effect of the extraction of j, allowing to shortcut nodes corresponding to k as the following:

$$\begin{tikzcd}
\cdots \arrow[r, "\text{val}"] & E(h) \arrow[r, "\text{pr}"] & R(b,t_D^j) \arrow[r, "\text{pr}"] & G(b_j,t_D^j)(j) \arrow[rrr, bend right, "\text{pr}"'] & R(b,\_) \arrow[r, "\text{pr}"] & G(b,t_D^k)(k) \arrow[r, "\text{pr}"] & G(b,t_D^x)(x) \arrow[r, "\text{lpr} \lor \text{pr}"] & \cdots
\end{tikzcd}$$

These three transformations can be applied again considering $E(h)$ and the subsequent read $R(b,\_)$ and so on until we obtain the following graph:

$$\begin{tikzcd}
D(a) \arrow[r, "\text{val}"] & E(a) \arrow[r, "\text{pr}"] & R(b,\_) \arrow[r, "\text{pr}"] & G(b,t_D^y)(y) \arrow[r, "\text{pr}"] & G(b,t_D^x)(x) \arrow[llll, bend left, "\text{pr}"]
\end{tikzcd}$$

in which the enqueue of a precedes a read of `current` implying that the extraction of a must have occurred in the history by Lemma 9 or 11. Thus, the edge from $G(b,t_D^x)(x)$ to $D(a)$ belongs to the precedence relation. Now, if we apply the last transformation again, we obtain a graph as the following:

$$\begin{tikzcd}
R(b_a,t_D^a) \arrow[r, "\text{pr}"] & G(b_a,t_D^a)(a) \arrow[r, dashed, "\text{pr}"] \arrow[rr, dashed, bend left, "\text{pr}"] & R(b,\_) \arrow[r, "\text{pr}"] & G(b,t_D^y)(y) \arrow[r, "\text{pr}"] & G(b,t_D^x)(x) \arrow[llll, bend left, "\text{pr}"]
\end{tikzcd}$$

in which only one of the two dashed edges appears, nodes correspond to atomic instructions (either reads of `current` or `CAS` for extracting an item) and it preserves the cycle. Moreover, this graph has edges that belong only to the *pr* relation with a cycle. However, since nodes correspond to linearizable (atomic) instructions, linearizability is a local property (closed under composition), and atomics are totally ordered w.r.t. the precedence relation, the existence of such a cycle is impossible, meaning that there is no cycle at all. □

In order to proceed and prove that the algorithm is correct when reshuffles are enabled, we first need to characterize them.

Definition 1. *A reshuffle phase:*

- *starts with the successful atomic operation at Line R3 of Algorithm 3, denoted as $\alpha(\mathsf{NULL}, N)$;*
- *ends with the successful atomic operation at Line R29 of Algorithm 3, denoted as $\omega(N', N)$.*

Thanks to this, we can prove that a new reshuffle phase cannot start until the previous one has been completed.

Lemma 13. *Reshuffle phases cannot overlap with each other.*

Proof. Suppose that k reshuffle phases overlap. This means that k occurrences of α are executed without being interleaved by any ω operation. Thus:

$$\cdots \prec \alpha(\mathsf{NULL}, N_{h-1}) \prec \alpha(\mathsf{NULL}, N_h) \prec \alpha(\mathsf{NULL}, N_{h+1}) \cdots$$

and each one succeeds. Since the old value is fixed to `NULL` in each invocation and new values are always different from `NULL`, this is possible if and only if the swapped memory cell has a different address for each `CAS`. It means that there is an instruction that updates the address of the current set table between each pair of consecutive `CAS` operations starting a reshuffle phase. By analyzing the code, we know that these are successful ω-`CAS` operations. It follows that each reshuffle phase is completed before a new one begins, contradicting the hypothesis. □

Lemma 14. *When a reshuffle ends, all the items in the old set table are either extracted or migrated to the new set table.*

Proof. From the proof of Lemma 2, we know that the number of items stored in the old set table is bounded by a constant and a reshuffle phase ends when the old set table is empty. Since an item is stored in up to one VAL node by Lemma 5, a reshuffle phase eventually ends by Lemma 2, and a node is removed from a PBQ if it is marked as DEL by algorithm construction, we know that all VAL items in the old set table have been marked as DEL during a reshuffle. Consequently, any VAL item in the old set table has been removed from a PBQ by extractions (applying the VAL ⟶ DEL transitions) and/or migrations (applying the VAL ⟶ MOV ⟶ DEL transition). □

Theorem 2. **NotOrd** *violations cannot happen for any history generated by running the LFCQ algorithm.*

Proof. Lemma 13 tells us that we can focus on a single occurrence of the reshuffle phase from a set table A to a set table B. Without loss of generality, let us assume that there is a **NotOrd** violation and that there is an extraction $D()(x)$ such that LRCG(x) has a cycle.

Suppose that the successful invocation of GetMin in $D()(x)$ completes before the beginning of the reshuffle phase. It follows that x has been extracted from a PBQ a belonging to A. Since the reshuffle code does not modify current in set table A, it means that $D()(x)$ has extracted x regardless of the reshuffle phase, namely by observing an internal state (current and the PBQ) never modified by the migration protocol. Consequently, it is possible to build a history with no reshuffle that leads to the same wrong extraction, contradicting Lemma 12.

Consider that the CAS within the successful invocation of GetMin occurs during a reshuffle. Since GetMin has successfully extracted x from a PBQ a of set table A, it means that no migration has affected such an extraction, otherwise GetMin should have returned MOV. In more detail, before migrating a node n, the resize algorithm first ensures that the successor of n is also marked as MOV. This guarantees that if a dequeue has extracted x, the relative GetMin could not have traversed any DEL node due to a migration in the VB. In fact, we know that before deleting a MOV item, its successor has been marked as MOV. This would end up in having a GetMin detect some node marked as MOV eventually, making the considered GetMin abort. Thus, similarly to the previous case, the last successful attempt of the interested dequeue has observed an internal state not yet updated by the reshuffle protocol. Consequently, it is possible to build a history without reshuffles that leads to the same wrong extraction, contradicting Lemma 12.

At this point, we need to consider the case where GetMin linearizes after the end of the reshuffle phase. When a reshuffle phase ends, all nodes have been migrated from the old table to the new one by Lemma 14. It follows that x has been extracted from a PBQ b belonging to B. We know that priorities in the left-right-constraint graph of x have either been inserted directly into B or migrated from A. Since current in the new table is updated to match the VB of an item before it is actually migrated (see lines R22-R26 of Algorithm 3), it is guaranteed that when a reshuffle completes, current points to the VB with the minimal key. It means that $D()(x)$ has extracted x regardless of the reshuffle phase. Consequently, it is possible to build a history without reshuffle operations leading to the same wrong extraction, contradicting Lemma 12. □

5.1.3 *Avoidance of the* **FalseEm** *violation.*

Theorem 3. *The* **FalseEm** *violation cannot occur in any execution obtained by running the LFCQ algorithm.*

Proof. Let us assume that a Dequeue invocation completes by returning ⊥, i.e., detecting the queue as empty, even though it was never logically empty during the operation. This means that the current set table has size equal to 1 (see line D15), no migration is occurring (the head node is not marked as MOV), and the traversal carried out by the respective GetMin has reached the tail node. The latter operation consists of a simple read instruction of the next field of the predecessor of the tail node. We denote such an operation as r. Consequently, we can consider such an atomic operation as a successful extraction of an item whose key has an infinite value T, i.e., all keys in the priority domain have lower values. It follows that we can apply the same reasoning used in Lemma 12 by building an LRCG for key T in which its GetMin is materialized in the atomic read r instead of an atomic CAS, which leads to a contradiction. It follows that the unique PBQ in the current set table is actually empty. Consequently, since **FalseEm** occurs by hypothesis, it means that some item inserted in an old set table before the last reshuffle has not been extracted. This is a contradiction, since Lemma 14 states that all the items are extracted and/or migrated from the old set into the new set table during reshuffles.

□

5.1.4 LFCQ linearizability.

Theorem 4. *The LFCQ implements the Set semantics.*

Proof. The LFCQ algorithm is free from **MultiDeq**, **MultiExt**, **NotEnq**, and **FalseEm** violations as proven by Lemmas 6, 7, 8 and Theorem 3, respectively. □

Theorem 5. *LFCQ algorithm is linearizable w.r.t. the priority queue specification.*

Proof. From Theorem 4 we know that LFCQ implements set semantics. Also, Theorem 2 guarantees that **NotOrd** violations do not occur. Hence, LFCQ implements the priority queue semantics. □

6 EXPERIMENTAL EVALUATION

The experimental evaluation of our LFCQ has been conducted on a machine equipped with an Intel Xeon Gold 6338, for a total of 64 hardware threads, running Linux Kernel 5.15.0 and glibc 2.35. The test is based on an implementation of the HOLD benchmark [29], where each thread continuously executes a hold operation, namely a dequeue followed by an enqueue, on a pre-populated queue. The benchmark runs for 10 seconds, and we used the average throughput of operations as the performance metric. Finally, the benchmark and implementation of the algorithms[2] that we compare in this study have been compiled with gcc 11.3.0 and the O3 optimization flag.

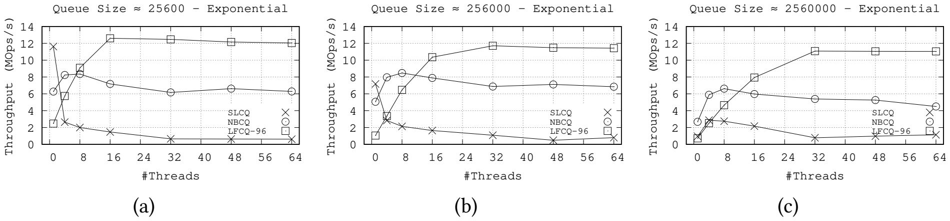
Initially, we have comparatively evaluated three different Calendar Queue (CQ) implementations. The first one, denoted SLCQ, is serialized via a spinlock. It simply consists of an implementation of the algorithm presented in [4], with the addition of the spinlock acquisition right before executing any operation. The spinlock is released as soon as the read and write operations on shared data implementing the calendar queue are completed. The second one is the Non-Blocking Calendar Queue (NBCQ) that has been presented in [23]. As we already noted, this algorithm has been devised to handle non-blocking (lock-free) concurrent operations. However, unlike the algorithm we present in this article, this solution does not offer any conflict resiliency of concurrent operations. Anyhow, we can consider NBCQ as the unique literature reference in terms of CQ algorithms offering the non-blocking property. Finally, the third solution is the implementation of the LFCQ algorithm we have presented.

In all of our experiments, the policy used to trigger the resizing of calendar queues is the one originally presented in [4]. In particular, the calendar queue is doubled (halved) in size when the number of items doubles (halves). Hence, for all algorithms, the number of resize operations occurring in a run with a given workload is the same.

[2] Available at https://github.com/HPDCS/LFCQ

Table 1. Distributions used in the evaluation tests.

Probability Distribution	Skewness	Formula
Uniform $(0, 2)$	0	$2 \cdot \mathtt{rand}$
Triangular $(a = 0, b = c = \frac{3}{2})$	$\frac{-2\sqrt{2}}{5}$	$\frac{3}{2} \cdot \sqrt{\mathtt{rand}}$
(Negative) Triangular $(a = b = 0, c = 3)$	$\frac{2\sqrt{2}}{5}$	$3 \cdot (1 - \sqrt{\mathtt{rand}})$
Exponential $(\lambda = 1)$	2	$\mathbf{if}(x \leftarrow \mathtt{rand} = 0)\ \infty\ \mathbf{else}\ -\ln(\mathtt{rand})$
Camel	0	(see [29])
Pareto $(x_m = \frac{3}{4}, \alpha = 4)$	$5\sqrt{2}$	$\frac{3}{4}(\mathtt{rand})^{-\frac{1}{4}}$



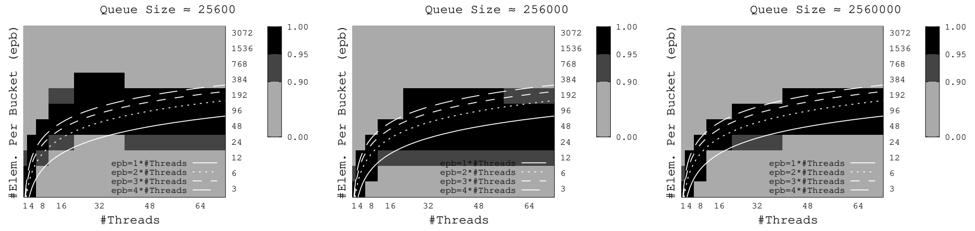
(a) (b) (c)

Fig. 8. Throughput of the hold benchmark with different calendar queue implementations: the original calendar queue serialized with a spinlock (SLCQ), NBCQ and LFCQ-96.

For both SLCQ and NBCQ, the resize leads to having three events per bucket on average. This value was initially suggested in [4] and has been adopted in [23] as the reference choice for NBCQ. For LFCQ, we present data with different choices concerning the number of events per bucket (EPB). As for this aspect, we note that in LFCQ, when the end of a virtual bucket is reached upon extractions, several tasks are performed, involving multiple atomic operations (physical removal of nodes or `current` updates). Hence, there is a relation between EPB and the frequency of these operations, which can be investigated by exploiting different choices for EPB. In an initial set of experiments, we configured LFCQ to maintain 96 items per bucket (denoted as LFCQ-96). From early data on the performance of LFCQ, we noted that this non-minimal value allows achieving high performance in scenarios with larger thread counts. This is related to the conflict resiliency offered by LFCQ. In fact, even if the operations last more clock cycles since the EPB to be scanned is longer, we do not observe adverse effects on performance caused by the increase of the effects of conflicts that lead to operation retries. However, to further optimize LFCQ in scenarios where the number of threads is lower, we also manage the EPB value by relying on an adaptive heuristic-based mechanism. The latter enables resizing EPB to reduce the impact of the aforementioned scan operations when this cost has a higher impact than executing actually conflicting atomic operations on the queue. Data related to this aspect are reported in the second part of this study.

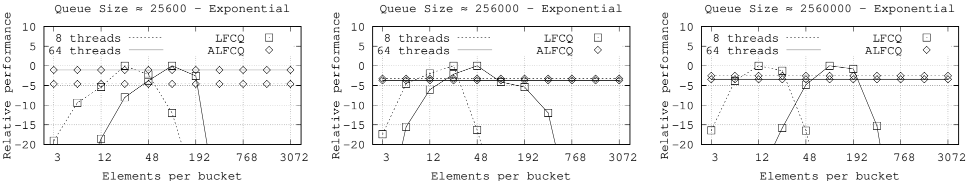
Figure 8 shows the average throughput while executing the benchmark with a priority increment following an exponential distribution[3] (shown in Table 1) and queue sizes varying in $\{256 \cdot 10^2,\ 256 \cdot 10^3,\ 256 \cdot 10^4\}$. Clearly, lock-free approaches pay off against SLCQ when the number of threads is greater than 1. At the same time, the NBCQ algorithm has a scalability collapse with more

[3] Each configuration's experiment has been executed 10 times and we report the average values over all the results.



(a) (b) (c)

Fig. 9. Performance analysis with varying queue sizes, number of events per bucket (EPB) and concurrency level. Each chart reports a heat map of the ratio between the performance of the given configuration and the optimum (the darker, the better) for a given queue size.



(a) (b) (c)

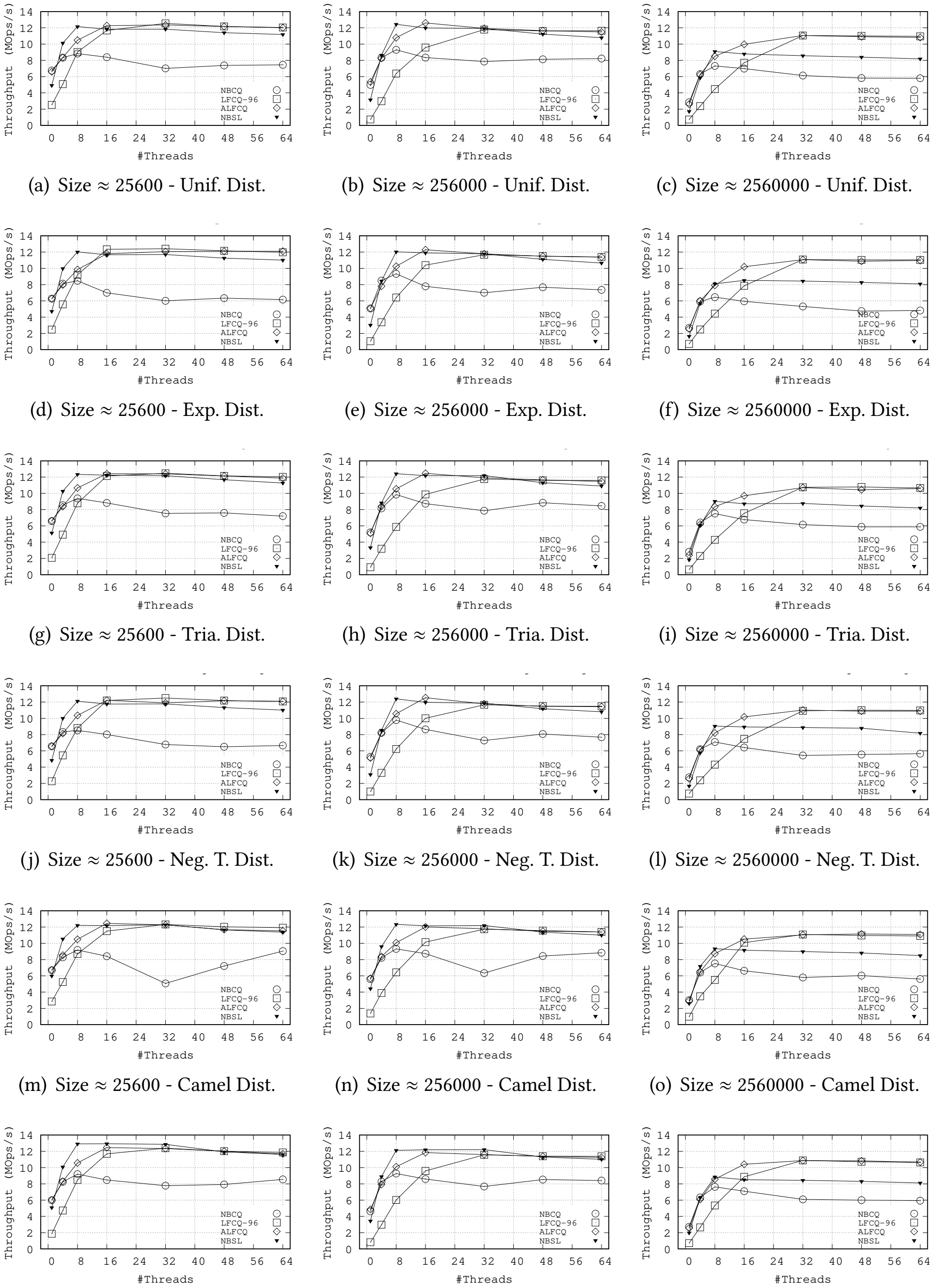
Fig. 10. Relative performance of ALFCQ w.r.t the optimal LFCQ configuration with 8 and 64 threads.

than eight threads, while LFCQ-96 maintains a higher and stable throughput, clearly showing the benefits it can provide.

As noted previously, the static value of 96 used for EPB in these experiments favours the effectiveness of conflict resiliency when we run with a higher number of threads. In the next part of this study, we investigate how to improve the performance of LFCQ at lower thread counts thanks to the dynamic selection of the suited EPB value and the optimization of the aforementioned trade-off between the latency of scan operations and the effect of atomic instructions.

This trade-off has been explored by evaluating the throughput while varying both EPB and the number of threads. Figure 9 shows the results with an exponential priority increment distribution. The charts are heat maps where each tile represents the ratio between the throughput of the current configuration (active threads and EPB) and the throughput obtained with the optimal bucket width for the target level of concurrency (number of threads). Light grey tiles are configurations that have a performance that is lower than 90% of the optimum, while dark grey and black tiles correspond to the values in [90%, 95%) and [95%, 100%] of the optimum, respectively. The results show that the higher the concurrency level, the higher the EPB should be to achieve throughput optimization. Also, the queue size has an impact on the optimal bucket width. In fact, large EPB values are convenient with small queues. However, choosing the EPB as a linear function of the number of active threads (T) is sufficient to model the behavior of dark tiles. In particular, choosing EPB $= N \cdot T$, where $N \in [1, 4]$, covers most of the region with a distance lower than 10% from the optimum. Consequently, we suggest configuring the LFCQ data structure by setting the EPB to about three times the number of active threads.

In the implementation of LFCQ that exploits this approach for the dynamic selection of EPB, which we denote as Adaptive LFCQ (ALFCQ), we have added an atomic counter for the number of completed operations, which is updated via FAD. Thanks to this, we can estimate the actual



(a) Size ≈ 25600 - Unif. Dist.

(b) Size ≈ 256000 - Unif. Dist.

(c) Size ≈ 2560000 - Unif. Dist.

(d) Size ≈ 25600 - Exp. Dist.

(e) Size ≈ 256000 - Exp. Dist.

(f) Size ≈ 2560000 - Exp. Dist.

(g) Size ≈ 25600 - Tria. Dist.

(h) Size ≈ 256000 - Tria. Dist.

(i) Size ≈ 2560000 - Tria. Dist.

(j) Size ≈ 25600 - Neg. T. Dist.

(k) Size ≈ 256000 - Neg. T. Dist.

(l) Size ≈ 2560000 - Neg. T. Dist.

(m) Size ≈ 25600 - Camel Dist.

(n) Size ≈ 256000 - Camel Dist.

(o) Size ≈ 2560000 - Camel Dist.

(p) Size ≈ 25600 - Pareto Dist.

(q) Size ≈ 256000 - Pareto Dist.

(r) Size ≈ 2560000 - Pareto Dist.

Fig. 11. Throughput of the hold benchmark with different priority queue implementations and distributions.

concurrency level of the data structure and use it during reshuffle phases. Figure 10 shows a comparison between LFCQ with a static selection of the EPB value and ALFCQ with 8 and 64 threads. On the one hand, ALFCQ configured with $N = 3$ chooses an EPB value that leads to a performance no more than 5% lower than the optimum. On the other hand, the static choice of EPB performed by LFCQ might not fit the actual concurrency level, leading to suboptimal performance.

Finally, we can see that ALFCQ always performs as the best of, or better than, LFCQ and NBCQ, having a throughput consistently higher than both, or at least equal to the best of them. Figure 11 shows that such a result is consistent in all the priority-increment distributions and queue sizes evaluated. Also, (A)LFCQ provides up to 10% performance improvement over NBSL—which is an implementation of a non-blocking skip list—for small queue sizes. The speedup increases to 1.30x for very large queue sizes and up to 1.6x with no concurrency.

7 CONCLUSIONS

In this work, we have presented the Lock-Free-Calendar-Queue (LFCQ) algorithm. It is a priority queue based on the calendar approach, which provides both constant-time access and lock-free progress of concurrent operations and is at the same time provably correct. LFCQ embeds new techniques that provide conflict resilience of operations by concurrent threads. This is an essential feature for concurrent priority queues. In fact, they are known to be challenging since their semantics makes extracting threads highly prone to conflicting execution paths, which may require to be re-executed after an abort. We have proved that our LFCQ algorithm is linearizable by means of an assertional approach, namely by showing that priority queue invariants are always preserved. Our correctness proof goes beyond the well-known approach typically exploited for proving correctness based on fixed linearization points.

We evaluated the effectiveness of the LFCQ algorithm, comparing it to both blocking and lock-free priority queues available in the literature. The experimental results, achieved running the hold benchmark on a machine with 64 CPUs, show how LFCQ can provide important performance boots, particularly in scenarios with larger queue sizes and higher concurrency levels.

REFERENCES

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A Scalable Relaxed Priority Queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP'15)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/2688500.2688523

[2] Rassul Ayani. 1990. LR-Algorithm: concurrent operations on priority queues. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing (SPDP)*. IEEE Computer Society, Dallas, TX, USA, 22–25.

[3] Ahmed Bouajjani, Constantin Enea, and Chao Wang. 2017. Checking Linearizability of Concurrent Priority Queues. In *Proceedings of the 28th International Conference on Concurrency Theory (Leibniz International Proceedings in Informatics, Vol. 85)*, Roland Meyer and Uwe Nestmann (Eds.). Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 16:1–16:16. https://doi.org/10.4230/LIPIcs.CONCUR.2017.16

[4] Randy Brown. 1988. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Commun. ACM* 31, 10 (1988), 1220–1227.

[5] Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2015. Aspect-oriented linearizability proofs. *Logical Methods in Computer Science* Volume 11, Issue 1 (April 2015), 1–33. https://doi.org/10.2168/LMCS-11(1:20)2015

[6] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2015. RAMSES: Reversibility-based agent modeling and simulation environment with speculation-support. In *Euro-Par 2015: Parallel Processing Workshops*, Sascha Hunold, Alexandru Costan, Domingo Giménez, Alexandru Iosup, Laura Ricci, María Engracia Gómez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander (Eds.). Lecture Notes in Computer Science, Vol. 9523. Springer International Publishing, Cham, Germany, 466–478. https://doi.org/10.1007/978-3-319-27308-2_38

[7] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. No Hot Spot Non-blocking Skip List. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Piscataway, NJ, USA, 196–205. https://doi.org/10.1109/ICDCS.2013.42

[8] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. 2018. NUMASK: High Performance Scalable Skip List for NUMA. In *Proceedings of the 32nd International Symposium on Distributed Computing (Leibniz International Proceedings in Informatics, Vol. 121)*, Ulrich Schmid and Josef Widder (Eds.). Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:19. https://doi.org/10.4230/LIPIcs.DISC.2018.18

[9] Brian C. Dean and Zachary H. Jones. 2007. Exploring the Duality Between Skip Lists and Binary Search Trees. In *Proceedings of the 45th Annual Southeast Regional Conference* (Winston-Salem, North Carolina) *(ACM-SE'07)*. ACM, New York, NY, USA, 395–399. https://doi.org/10.1145/1233341.1233413

[10] Ian Dick, Alan Fekete, and Vincent Gramoli. 2017. A skip list for multicore. *Concurrency and Computation: Practice & Experience* 29, 4 (Jan. 2017), e3876. https://doi.org/10.1002/cpe.3876

[11] Mike Dodds, Andreas Haas, and Christoph M. Kirsch. 2015. A Scalable, Correct Time-Stamped Stack. *SIGPLAN Not.* 50, 1 (Jan. 2015), 233–246. https://doi.org/10.1145/2775051.2676963

[12] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge.

[13] Sounak Gupta and Philip A. Wilsey. 2014. Lock-free Pending Event Set Management in Time Warp. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Denver, Colorado, USA) *(SIGSIM PADS '14)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2601381.2601393

[14] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC'01)*. Springer-Verlag, London, UK, 300–314. http://dl.acm.org/citation.cfm?id=645958.676105

[15] Joshua Hay and Philip A. Wilsey. 2015. Experiments with Hardware-based Transactional Memory in Parallel Simulation. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (London, United Kingdom) *(SIGSIM PADS '15)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/2769458.2769462

[16] Maurice Herlihy. 1991. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems* 13, 1 (Jan. 1991), 124–149. https://doi.org/10.1145/114005.102808

[17] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[18] Maurice Herlihy and Nir Shavit. 2011. On the Nature of Progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems* (Toulouse, France) *(OPODIS'11)*. Springer-Verlag, Berlin, Heidelberg, 313–328. https://doi.org/10.1007/978-3-642-25873-2_22

[19] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[20] Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2018. The Ultimate Share-Everything PDES System. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Rome, Italy) *(SIGSIM-PADS '18)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/3200921.3200931

[21] Clyde P Kruskal, Larry Rudolph, and Marc Snir. 1988. Efficient synchronization of multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems* 10, 4 (1988), 579–601.

[22] Jonatan Lindén and Bengt Jonsson. 2013. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *Principles of Distributed Systems*, Roberto Baldoni, Nicolas Nisse, and Maarten van Steen (Eds.). Springer International Publishing, Cham, Germany, 206–220.

[23] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2016. A Lock-Free O(1) Event Pool and Its Application to Share-Everything PDES Platforms. In *Proceedings of the 20th International Symposium on Distributed Simulation and Real-Time Applications* (Uxbridge, United Kingdom) *(DS-RT'16)*. IEEE Press, Piscataway, NJ, USA, 53–60. https://doi.org/10.1109/DS-RT.2016.33

[24] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2016. A Non-Blocking Priority Queue for the Pending Event Set. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques* (Prague, Czech Republic) *(SIMUTOOLS'16)*. ICST, Brussels, Belgium, Belgium, 46–55. http://dl.acm.org/citation.cfm?id=3021426.3021434

[25] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2017. A Conflict-Resilient Lock-Free Calendar Queue for Scalable Share-Everything PDES Platforms. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Singapore) *(SIGSIM-PADS'17)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/3064911.3064926

[26] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. https://doi.org/10.1145/78973.78977

[27] Francesco Quaglia. 2015. A low-overhead constant-time Lowest-Timestamp-First CPU scheduler for high-performance optimistic simulation platforms. *Simulation Modelling Practice and Theory* 53 (2015), 103 – 122. https://doi.org/10.1016/j.simpat.2015.01.009

[28] Dhananjai M. Rao and Julius D. Higiro. 2019. Managing Pending Events in Sequential and Parallel Simulations Using Three-tier Heap and Two-tier Ladder Queue. *ACM Transactions on Modeling and Computer Simulation* 29, 2, Article 9

(March 2019), 28 pages. https://doi.org/10.1145/3265750

[29] Robert Rönngren and Rassul Ayani. 1997. A Comparative Study of Parallel and Sequential Priority Queue Algorithms. *ACM Transactions on Modeling and Computer Simulation* 7, 2 (April 1997), 157–209. https://doi.org/10.1145/249204.249205

[30] Emanuele Santini, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2015. Hardware-Transactional-Memory Based Speculative Parallel Discrete Event Simulation of Very Fine Grain Models. In *Proceedings of the 22nd International Conference on High Performance Computing (HiPC)*. IEEE, Piscataway, NJ, USA, 145–154. https://doi.org/10.1109/HiPC.2015.45

[31] N Shavit and I Lotan. 2000. Skiplist-based concurrent priority queues. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Piscataway, NJ, USA, 263–268. https://doi.org/10.1109/IPDPS.2000.845994

[32] Håkan Sundell and Philippas Tsigas. 2005. Fast and Lock-free Concurrent Priority Queues for Multi-thread Systems. *J. Parallel Distrib. Comput.* 65, 5 (May 2005), 609–627. https://doi.org/10.1016/j.jpdc.2004.12.005

[33] Wai Teng Tang, Rick Siow Mong Goh, and Ian Li-Jin Thng. 2005. Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation. *ACM Transactions on Modeling and Computer Simulation* 15, 3 (July 2005), 175–204. https://doi.org/10.1145/1103323.1103324