BQ: A Lock-Free Queue with Batching*

GAL MILMAN, Technion, Israel
ALEX KOGAN, Oracle Labs, USA
YOSSI LEV, Oracle Labs, USA
VICTOR LUCHANGCO, Oracle Labs, USA
EREZ PETRANK, Technion, Israel

Concurrent data structures provide fundamental building blocks for concurrent programming. Standard concurrent data structures may be extended by allowing a sequence of operations to be submitted as a batch for later execution. A sequence of such operations can then be executed more efficiently than the standard execution of one operation at a time. In this paper we develop a novel algorithmic extension to the prevalent FIFO queue data structure that exploits such batching scenarios. An implementation in C++ on a multicore demonstrates a significant performance improvement of up to 16x (depending on the batch lengths and the number of threads), compared to previous queue implementations.

Additional Key Words and Phrases: Concurrent Algorithms; Concurrent Data Structures; Lock-Freedom; Linearizability; FIFO Queue

1 INTRODUCTION

The era of multi-core architectures has been having a huge impact on software development: exploiting concurrency has become the main challenge of today's programming. Concurrent data structures provide the basic blocks for concurrent programming; hence it is crucial that they are efficient and scalable. In this paper we consider a setting in which threads sometimes execute a *sequence* of operations on a shared concurrent data structure (rather than a single operation each time). This scenario occurs either because the threads are willing to delay execution of operations in order to improve performance, or because they deliberately want operations to be executed later.

As an example, consider a server thread that serves requests of remote clients. Such a thread may accumulate several relevant operations required by some client, generate a sequence of these operations, submit them for execution on shared data, finish handling them, and then proceed to handle other clients whose operations can be accumulated similarly.

Kogan and Herlihy [17] formulated batching of operations in a concurrent setting using the *future* programming construct. *Batching* means grouping a sequence of standard operations to a single *batch operation*, which applies them together to the shared object. They formalized correctness (linearization) guarantees and demonstrated the advantages of batching even when using naive batching strategies.

In this paper we present a novel extension to the concurrent lock-free queue by Michael and Scott [22] (henceforth MSQ) that can handle a sequence of operations in a batch. Our queue extension, denoted BQ (which stands for Batching Queue), provides faster execution for operation sequences. Kogan and Herlihy suggested to apply each sequence of operations of the same type to the shared queue at once. Specifically, they execute each subsequence of enqueues-only together by appending adequate nodes at the end of the queue, and each subsequence of dequeues-only by unlinking several nodes from the head of the queue. The advantage of this method degrades when operations in the batch switch frequently between enqueues and dequeues, which is the case with general sequences. We present an algorithm that handles any batch of enqueues and dequeues locally and applies it at once to the shared queue to reduce contention. Using novel observations on the effect

*This work was supported by the Israel Science Foundation grant No. 274/14

of a mixed sequence on the shared queue, we achieve a fast application of the sequence on the shared queue with low synchronization.

Concurrent queues are typically not scalable because they have two points of contention: the head and the tail. However, batching of operations provides an excellent opportunity to combine operations locally and improve scalability. Such local computation reduces the number of accesses to the shared structure, which yields an overall reduced contention. As shown in the measurements, BQ improves the performance and scalability over MSQ and over the simpler batching method of Kogan and Herlihy.

We also extend Kogan and Herlihy's formal treatment of systems that only execute batch operations, to allow simultaneous execution of standard (single) operations, while still satisfying an extended form of linearizability that we present.

BQ is lock-free. It uses only compare-and-swap (`CAS`) atomic operations (which can easily be replaced with *LL*/*SC* instructions) and can thus be ported to other existing platforms. The original MSQ we build upon is widely known as a well-performing queue for general hardware and is included as part of the Java™ Concurrency Package [20]. Measurements for BQ demonstrate a significant performance improvement of up to 16x compared to MSQ when threads employ batch operations to update the queue.

Batching provides a performance improvement for operations that the user agrees to delay. Additionally, BQ guarantees that deferred operations of a certain thread will not take effect until that thread performs a non-deferred operation or explicitly requests an evaluation of previous future operations. This is useful when the user wishes to call several operations and knowingly delay their execution to a chosen time.

The rest of the paper is organized as follows: Section 2 introduces the model we work with and surveys the work we build on. In Section 3 we define linearizability and its extensions to objects with batch operations. Having set the terminology, we discuss related work in Section 4. Section 5 presents an overview of the BQ algorithm, whose implementation details are described in Section 6. The memory management mechanism we used is covered in Section 7. The algorithm's correctness is laid out in Section 8. Section 9 describes measurement results. Section 10 lays out a possible portability adjustment of the algorithm.

2 PRELIMINARIES

Model. We consider a standard shared memory setting, with a set of threads accessing the shared memory using the atomic primitives `Read`, `Write` and `CAS`. A `CAS` primitive is defined by a triplet consisting of a target memory address, an expected value, and a new value. A `CAS` operation compares the value stored in the target address to the expected value. If they are equal, the value in the target address is replaced with the new value, and the Boolean value true is returned. In such a case we say that the `CAS` is successful. Otherwise, the shared memory remains unchanged, and false is returned.

Future. A *future* is an object returned by an operation whose execution might be delayed. The user may call an *Evaluate* method to ensure the operation's execution and get its result.

MS-Queue. BQ extends MSQ to support future operations. MSQ is a lock-free algorithm for a FIFO queue, which supports *Enqueue* and *Dequeue* operations. It implements the queue as a singly-linked list with *head* and *tail* pointers. *head* points to the first node of the list, which functions as a dummy node. The following nodes, starting with the node pointed to by the dummy node's *next* pointer and ending with the node whose *next* pointer's value is `NULL`, contain the queue's items. The queue is initialized as a list containing a single (dummy) node, to which both *head* and *tail* point. This setup represents an empty queue.

Dequeuing is implemented as follows: If *head->next* is `NULL`, the queue is empty, and hence the dequeue operation returns without extracting an item from the queue. Otherwise, an attempt is made to update *head* to point to its successive node in the list, using `CAS`. On the occasion that the `CAS` fails, the dequeue operation starts over.

Enqueuing requires two `CAS` operations. Initially, a node with the item to enqueue is created. Then, an attempt to set *tail->next* to the address of the new node is made using a first `CAS`. The `CAS` fails if the current value of *tail->next* is not `NULL`. In such a case, *tail* is advanced to the current value of *tail->next* using an assisting `CAS`, in order to help an obstructing enqueue operation complete. Then, a new attempt to perform the first `CAS` starts. After the first `CAS` succeeds, a second `CAS` is applied to update *tail* to point to the new node. There is no need to retry this `CAS`, since it fails only if another thread has already performed the required update, trying to help our operation complete in order to next apply its own operation.

Lock-Freedom. A concurrent object implementation is *lock-free* [12] if each time a thread executes an operation on the object, some thread (not necessarily the same one) completes an operation on the object within a finite number of steps. Thus, lock-freedom guarantees system-wide progress. Our implementation is lock-free.

3 LINEARIZABILITY AND FUTURES

We describe the original linearizability [14], defined for a setting of no future operations, and generalize it for a setting with future operations. Some basic terms are required first: A *method call* is described by two events – its *invocation*, which refers to the call to the method, and its *response*, which refers to the return from the method. Each object has a *sequential specification*, which describes its behavior in sequential executions, where method calls do not overlap.

3.1 Linearizability

An execution is considered *linearizable* [14] if each method call appears to take effect at once, between its invocation and its response events, in a way that satisfies the sequential specification of the objects.

3.2 Medium Futures Linearizability (MF-Linearizability)

Medium futures linearizability is defined by Kogan and Herlihy [17] as an extension of linearizability to futures, which we adopt and extend. For each future operation, we look at two associated method calls: the future one, which creates a future and returns it, and *Evaluate*, which is called with the future returned by the first method call and ensures the operation's execution. MF-linearizability requires the following:

(1) Each operation takes effect at some instant between the invocation of its first related method (which produces the future) and the response of its second related method (which evaluates the future).
(2) Two operations issued by the same thread to the same object take effect in the order of their first method calls (i.e. their future method calls).

3.3 Extended Medium Futures Linearizability (EMF-Linearizability)

We extend the MF-linearizability definition to cover a data structure which supplies standard (single) operations in addition to future-returning operations (the original paper did not refer to single operations). We do so by reduction to MF-linearizability: informally, we transform an execution that possibly contains single calls into one that contains only future-returning operations, by replacing each single operation call with an adequate future call followed by an *Evaluate* call.

Next, we define *extended medium futures linearizability* (*EMF-linearizability*) formally.

Definition 3.1. Let H be a history, consisting of operation invocation and response events. We construct a new history H_f, denoted the *future history*, as follows. The invocations and responses of all future dequeue, future enqueue and *Evaluate* calls are copied to H_f unchanged. Each dequeue call *op* in H is rewritten in H_f: its invocation is replaced with an invocation and an immediate response of a future dequeue call, and its response is replaced with an invocation and an immediate response of an *Evaluate* call that evaluates this future dequeue. Enqueue calls are rewritten similarly.

Definition 3.2. A history H is *EMF-linearizable* if its future history H_f is MF-linearizable.

3.4 Atomic Execution

We define *atomic execution*, a property of an object with future methods regarding its linearization. We then describe its implication for EMF-linearizable objects and its benefit. We begin with an auxiliary term regarding future operations on shared objects:

Definition 3.3. A *pending operation* is a future operation that has not yet been applied to the shared object.

Definition 3.4. Let *ob* be an object with future methods and t be a thread. *ob* satisfies *atomic execution* if:

(1) Pending operations of t on *ob* are applied only during the following calls on *ob* by t: either a single operation call, or an *Evaluate* call for one of t's pending operations.
(2) Let *op* be a call in t during which some pending operations of t, denoted $op_1, ..., op_n$, are applied. Then $op_1, ..., op_n$ must be linearized successively in their original invocation order, without any other operation on *ob* linearized between them. Moreover, if *op* is a single operation (i.e., not an *Evaluate* or a future operation), then $op_1, ..., op_n$ and also *op* must be linearized successively in their original invocation order, without any other operation on *ob* linearized between them.

If *ob* is EMF-linearizable, then when a single method *op* is called by a thread t, all pending operations of t must take effect prior to *op* to satisfy EMF-linearizability. Atomic execution dictates that they are executed at once together with *op*.

Similarly, when t calls the *Evaluate* method for some pending operation *op*, all t's pending operations preceding *op* must take effect prior to *op* to satisfy EMF-linearizability. Additional pending operations by t may be applied as well. Atomic execution dictates that all these operations are executed at once.

An example of a potential benefit of atomic execution is that it can achieve locality for a producers-consumers system, where consumer servers handle requests of remote producer clients. In such scenario, the clients enqueue their requests, possibly several at a time, to a shared queue. Each server consumes requests regularly by performing a batch operation consisting of a certain number of dequeues. Both clients and servers perform batch operations by calling several future operations, followed by an *Evaluate* operation of the last future operation.

Serving requests of the same client consecutively may allow for more efficient processing due to locality of the client's data. A queue that supports batching and satisfies atomic execution would enable the servers to exploit locality and successively serve several requests by the same client, which he applied in the same batch-of-enqueues operation. This is thanks to the atomic execution's guarantee that both a batch-of-enqueues operation by a client and a batch-of-dequeues operation by a server take effect instantaneously.

4 RELATED WORK

Various papers introduce lock-free linearizable FIFO queues, which use different strategies to outperform MSQ.

Tsigas et al. [26] present a queue that allows the head and tail to lag at most m nodes behind the actual head and tail of the queue, so that the amortized number of CAS executions per operation is $1 + 1/m$. Their algorithm is limited to bounded queues due to their static allocation. Additional cyclic array queues are described in [3, 6, 25]. Moir et al. [23] present a queue that uses elimination as a back-off strategy to increase scalability: pairs of concurrent enqueue and dequeue method calls may exchange values without accessing the shared queue. However, in order to keep the FIFO queue semantics, the enqueue method can be eliminated only after all items of preceding enqueue operations have been dequeued, which makes the algorithm practical only for nearly empty or highly overloaded queues. Hoffman et al. [15] present the baskets queue, which increases scalability by allowing concurrent enqueue operations to insert nodes at adjacent positions at the end of the linked list, regarded as baskets. Such insertion, however, is done only after a failed initial attempt to append the node to the tail. Thus, the contention on the tail is only partially diminished, and there is also contention on the baskets.

Ladan-Mozes et al. [19] present an optimistic queue, which replaces one of the two CAS operations performed during an enqueue operation with simple stores. Like the original MSQ, this algorithm does not scale, due to synchronization on the head and tail variables that allows only one enqueue operation and one dequeue operation to be applied concurrently. Gidenstam et al. [7] present a cache-aware queue that stores the items in fixed-size blocks, connected in a linked list. This allows for a lazy update of the head and tail, only once per block. Nevertheless, at least one CAS per operation is still required, making the queue non-scalable under high contention. Morrison et al. [24] present a queue based on a linked list of ring queue nodes. To reduce contention, it relies on the fetch-and-add primitive to spread threads among items in the queue and let them operate in parallel. Yang et al. [27] utilize fetch-and-add as well, to form a wait-free queue. Queues that improve scalability by relaxing the sequential specification of the queue appear in [1, 10, 16]. For example, Basin et al. [1] suggest to trade fairness for lower contention by relaxing the FIFO semantics of the queue. The extension of linearizability BQ adheres to could be viewed as a relaxation, but a stricter one, as it forces FIFO semantics and preserves process order.

Previous works [e.g., 4, 5, 8, 9, 11, 13, 18] present concurrent constructs that combine multiple operations into a single operation on the shared object. We chose to combine operations and apply them as batches, in order to increase scalability. The paper of Kogan and Herlihy [17] is the closest to this work. They propose alternative definitions for linearizability of executions with batches, including MF-linearizability, which we use. They describe very simple implementations of stacks, queues and linked lists that demonstrate the benefits of using futures. In this work we propose a novel implementation of the queue that obtains better scalability and performance. Moreover, BQ satisfies atomic execution, while Kogan and Herlihy's simple queue does not.

5 ALGORITHM OVERVIEW

We present BQ, an extension to MSQ, which supports deferred operations and satisfies EMF-linearizability. Unlike standard operations, deferred operations need not be applied to the shared queue before their responses occur. When a future method is called, its details are recorded locally together with previous deferred operations that were called by the same thread. A *Future* object is returned to the caller, who may evaluate it later.

Deferred operations allow to apply pending operations in batches: BQ delays their execution until the user explicitly evaluates a future of one of them or calls a standard method. When that

happens, all pending operations of the same thread are gathered to a single batch operation. This operation is then applied to the shared queue. Afterwards, the batch execution is finalized by locally filling the futures' return values. This mechanism reduces synchronization overhead by allowing fewer accesses to the shared queue, as well as less processing in the shared queue – thanks to the preparations performed locally by the initiating thread during the run of each future operation, and the local pairing of applied futures with results following the batch execution.

5.1 Batch Execution

Whenever a deferred enqueue operation is called, the executing thread appends its item to a local list. This way, when the thread has to perform a batch operation, the list of nodes to be linked to the shared queue's list is already prepared.

The key to applying all operations of a batch at once to the shared queue, is to set up a moment in which the state of the queue is "frozen". Namely, we establish a moment in which we hold both ends of the queue, so that we know its head and tail, and its size right before the batch takes effect. This way we can unambiguously determine the queue's shape after applying the batch, including its new head and tail. We achieve a hold of the queue's ends by executing a batch operation in stages, according to the following scheme.

The thread first creates an announcement describing the required batch operation. An announcement is an auxiliary object used to announce an ongoing batch operation, so that other threads will not interfere with it but rather help it complete. Then, the thread modifies the shared queue's head to point to the created announcement. This marks the head so that further attempted dequeues will help the batch execution to be completed before executing their own operations. Now we hold one end of the queue.

Next, the initiating thread or an assisting thread links the list of items, which the initiating thread has prepared in advance, after the shared queue's tail. This determines the tail location after which the batch's items are enqueued. Thus, now we hold both ends of the queue, as required. We then update the shared queue's tail to point to the last linked node.

As a last step that would uninstall the announcement and finish the batch execution, we update the shared queue's head. It is possible that during the execution of the required enqueues and dequeues the queue becomes empty and that some of the dequeues operate on an empty queue and return `NULL`. We make a combinatorial observation that helps quickly determine the number of non-successful dequeues. This number is used to determine the node to which the queue's head points following the batch execution. By applying this fast calculation, we execute the batch with minimal interference with the shared queue, thus reducing contention. This computation is described in Section 5.2 below.

The entire algorithm, including the process of setting futures' results, is discussed in detail in Subsection 6.2.

5.2 A Key Combinatorial Property of Batches on Queues

Let us state combinatorial observations that help us execute the local batch quickly on the queue. The enqueued items in the batch are kept as a linked-list so that they can be attached at the end of the list in a single `CAS`. This list is added to the tail of the queue and then $\#successfulDequeues$ dequeues are executed by pushing the head $\#successfulDequeues$ nodes further in the shared linked-list representing the queue, and then dequeued items are privately matched with the batch dequeue operations. In the simplest scenario, $\#successfulDequeues$ equals the number of future dequeues in the batch. The problem is that some dequeues may operate on an empty queue and thus, must return a `NULL` value. The following discussion explains how the adequate $\#successfulDequeues$ can be computed.

Definition 5.1. We call a future dequeue *a failing dequeue* with respect to a given state of the shared queue, if the application of the batch that contains it (as well as the other local pending operations) on this shared queue makes this dequeue operate on an empty shared queue. A future dequeue that is not failing is called a *successful dequeue.*

Note that a failing dequeue does not modify the queue, and its future's result is `NULL`.

We start by analyzing the execution of a batch on an empty queue (which can be analyzed independently of the current shape of the shared queue) and then we show that this analysis can be extended to a general shared queue, simply by plugging the shared queue size.

Definition 5.2. An *excess dequeue* is a future dequeue operation that is a failing dequeue with respect to an empty queue.

For example, if the sequence of pending operations in some thread is EDDEEDDDEDDEE, where E and D represent enqueue and dequeue operations respectively, then the thread has three excess dequeues (the second, fifth and seventh).

An excess dequeue is a special case of a failing dequeue. We start by computing how many excess dequeues there are in a batch.

Lemma 5.3. *Let B be a batch of queue operations. The number of excess dequeues in this batch equals the maximum over all prefixes of this batch, of the number of dequeues in the prefix minus the number of enqueues in this prefix.*

Proof. First, we note that if, for some prefix p of the batch operations, the number of dequeues minus the number of enqueues is k, then the overall number of excess dequeues must be at least k. This is simply because when executing the prefix p on the empty queue, the number of items that enter the queue is $\#enqueues$, the number of enqueues in the prefix. On the other hand, $\#dequeues$, the number of dequeues that are executed in this prefix, is larger by k. So at least k dequeues must operate on an empty queue (returning `NULL`).

On the other hand, we show by induction on the number of excess dequeues that in the prefix that ends in the ℓ^{th} excess dequeue, $\#dequeues - \#enqueues \geq \ell$. We inspect the execution of the prefix on an empty queue. The base of the induction follows from the fact that the first excess dequeue must happen when the number of dequeues so far exceeds the number of enqueues. (Otherwise, there is an item to dequeue.) For the induction step we look at the prefix of the batch that ends in the $\ell - 1^{st}$ excess dequeue. By the induction hypothesis, for that prefix $\#dequeues - \#enqueues \geq \ell - 1$. Also, the queue must be empty after (any excess dequeue and in particular after) the $\ell - 1^{st}$ excess dequeue. So the subsequence of operations between the $\ell - 1^{st}$ excess dequeue and the ℓ^{th} excess dequeue operates on an empty queue and has an excess dequeue at the end, which means that for this subsequence $\#dequeues - \#enqueues \geq 1$ (like in the base case of the induction), and we are done. □

Now we proceed to discuss a batch applied to a queue of any size.

Claim 5.4. *Let n be the size of the queue right before a given batch is operated on it. The number of failing dequeues in the batch with respect to a queue of size n equals to the maximum value of $(\#dequeues - \#enqueues - n)$ over all prefixes of the batch's operation sequence, or 0 if this maximum is negative.*

The claim can be proven by adjusting the proof of Lemma 5.3 to failing dequeues instead of excess dequeues, and to a queue of general size n rather than 0. Note that the first n excess dequeues are not failing dequeues because they can dequeue the n items in the original queue. Any additional excess dequeues will become failing dequeues.

Claim 5.4 and Lemma 5.3 yield the following corollary:

Corollary 5.5. *Let n be the size of the queue right before a given batch is operated on it. The number of failing dequeues in the batch equals to* $\max\{\#excessDequeues - n, 0\}$.

It immediately follows that the number of successful dequeues in a batch with respect to a queue of size n equals:
$\#successfulDequeues = \#dequeues - \max\{\#excessDequeues - n, 0\}$

5.2.1 Using The Combinatorial Property in BQ. In order to optimize the calculation of the new head after a batch is applied, each thread maintains three local counters: the quantities of *FutureEnqueue* and *FutureDequeue* operations that have been called but not yet executed on the shared queue, and the number of excess dequeues. The thread updates these counters on each of its future operation calls. When a thread executes a batch operation, it includes its local counters in the batch's announcement, to allow any helping thread to complete the batch execution.

In addition, we let the shared queue's head and tail contain not only a pointer, but also a successful dequeue and enqueue counters respectively. When applying a batch, they are updated using the announcement's counts. The difference between the queue's enqueue and dequeue counts prior to a batch execution yields the queue's size n in its "frozen" state right before linking the batch's items.

These counters in the batch's announcement and in the head and tail are used to quickly calculate the number of successful dequeues according to Corollary 5.5. This number helps discovering the new head – by iterating over $\#successfulDequeues$ nodes, and avoids a heavier simulation of the batch enqueues and dequeues one by one to discover the shape of the resulting shared queue.

Indeed, to determine the result of each future dequeue in the batch, the thread that initiated the batch operation will need to simulate these future operations according to their order. Nevertheless, it will conduct this simulation after the announcement is removed from the shared queue, without delaying other threads that access the shared queue.

6 ALGORITHM DETAILS

We now turn to the details of the algorithm. In Section 6.1 we elaborate on the principal underlying data structures, and in Subsection 6.2 we describe the algorithm. Memory management is covered in Section 7.

6.1 Underlying Data Structures

Table 1 depicts the auxiliary data structures, out of which the *Future* structure is the only one exposed to the user of the *Queue* object, while all others are internal to the queue's implementation.

6.1.1 Queue. Similarly to MSQ, the shared queue is represented as a linked list of nodes in which the first node is a dummy node, and all nodes thereafter contain the values in the queue in the order they were enqueued. We maintain pointers to the first and last nodes of this list, denoted *SQHead* and *SQTail* respectively (which stand for *Shared Queue's Head and Tail*).

Batch operations require the size of the queue for a fast calculation of the new head after applying the batch's operations. To this end, *Queue* maintains counters of the number of enqueues and the number of successful dequeues applied so far. These are kept size-by-side with the tail and head pointers respectively, and are updated atomically with the respective pointers using a double-width `CAS`. This is implemented using a Pointer and Count object (*PtrCnt*, that can be atomically modified) for the head and tail of the shared queue.

Batch operations that enqueue at least one item install an announcement in the head. *SQHead* can either hold the aforesaid *PtrCnt* object with a pointer to the head of the queue, or a pointer to an *Ann* object, described next. Therefore, *SQHead* is a Pointer and Count or Announcement object

Table 1. Underlying Data Structures

```
struct Future          {result: Item*, isDone: Boolean}
struct Node            {item: Item*, next: Node*}
struct BatchRequest    {firstEnq: Node*,
                        lastEnq: Node*,
                        enqsNum: unsigned int,
                        deqsNum: unsigned int,
                        excessDeqsNum: unsigned int}
struct PtrCnt          {node: Node*,
                        cnt: unsigned int}
struct Ann             {batchReq: BatchRequest,
                        oldHead: PtrCnt,
                        oldTail: PtrCnt}
union PtrCntOrAnn      {ptrCnt: PtrCnt,
                        struct {tag: unsigned int,
                                ann: Ann*}}
struct FutureOp        {type: {ENQ, DEQ},
                        future: Future*}
struct ThreadData      {opsQueue: Queue of FutureOp,
                        enqsHead: Node*,
                        enqsTail: Node*,
                        enqsNum: unsigned int,
                        deqsNum: unsigned int,
                        excessDeqsNum: unsigned int}
```

(*PtrCntOrAnn*), which is a 16-byte union that may consist of either *PtrCnt* or an 8-byte tag and an 8-byte *Ann* pointer. Whenever it contains an *Ann*, the tag is set to 1. Otherwise, *SQHead* contains a *PtrCnt* (the tag overlaps *PtrCnt.node*, whose least significant bit is 0 since it stores either `NULL` or an aligned address).

For brevity, we will mostly avoid specific mention of the counter; however, when we refer to an update of the head's pointer, it means that the head's counter is updated as well, and likewise for the tail.

It is possible to avoid the double-width `CAS` in platforms that do not support such an operation. This can be accomplished by replacing the *PtrCnt* object with a pointer to a node, replacing the *PtrCntOrAnn* object with a pointer to either a node or an announcement (with a least significant bit mark indicating the type of the pointed object), and have the *Node* object contain a counter. We describe this variation of BQ in Section 10. Measurements demonstrate that it does not incur a significant performance degradation.

Thread-Local Data. A *threadData* array holds local data for each thread. First, the pending operations details are kept, in the order they were called, in an operation queue *opsQueue*, implemented as a simple local non-thread-safe queue. It contains *FutureOp* items. Second, the items of the pending enqueue operations are kept in a linked list in the order they were enqueued by *FutureEnqueue* calls. This list is referenced by *enqsHead* and *enqsTail* (with no dummy nodes here). Lastly, each thread keeps record of the number of *FutureEnqueue* and *FutureDequeue* operations that have been called but not yet applied, and the number of excess dequeues.

Each thread can access its local data in *threadData* using its thread ID as an index. In the pseudo-code, *threadData*[*threadId*] is abbreviated to *threadData* for brevity.

6.1.2 Future. A *Future* contains a *result*, which holds the return value of the deferred operation that generated the future (for dequeues only, as enqueue operations have no return value) and an *isDone* Boolean value, which is true only if the deferred computation has been completed. When *isDone* is false, the contents of *result* may be arbitrary.

6.1.3 BatchRequest. A *BatchRequest* is prepared by a thread that initiates a batch, and consists of the details of the batch's pending operations: $firstEnq$ and *lastEnq* are pointers to the first and last nodes of a linked list containing the pending items to be enqueued; $enqsNum$, $deqsNum$, and $excessDeqsNum$ are, respectively, the numbers of enqueues, dequeues and excess dequeues in the batch.

6.1.4 Announcement. An *Ann* object represents an announcement. It contains a *BatchRequest* instance, with all the details required to execute the batch operation it stands for. Thus, any operation that encounters an announcement may help the related batch operation complete before proceeding with its own operation.

In addition to information regarding the batch of operations to execute, *Ann* includes *oldHead*, the value of the head pointer (and dequeue counter) before the announcement was installed, and *oldTail*, an entry for the tail pointer (and enqueue counter) of the queue right before the batch is applied (i.e., a pointer to the node to which the batch's list of items is linked).

6.2 Algorithm Implementation

We detail the algorithm implementation, accompanied by pseudo-code. First we describe the core operations, performed on the shared queue. Then we outline the enclosing methods, which call the core methods and carry out complementary local computations. Finally we refer to the special case of a dequeues-only batch operation.

6.2.1 Internal Methods Operating on the Shared Queue. The following methods are used internally to apply operations to the shared queue: *EnqueueToShared*, *DequeueFromShared* and *ExecuteBatch*. To help a concurrent batch execution and obtain the new head, they call the *HelpAnnAndGetHead* auxiliary method. To carry out a batch, the *ExecuteAnn* auxiliary method is called. Its caller may be either the batch's initiating thread, or a helping thread that encountered an announcement when trying to execute its own operation.

Let us elaborate on each of these methods.

EnqueueToShared. EnqueueToShared appends an item after the tail of the shared queue, using two `CAS` operations, in a similar manner to MSQ's *Enqueue*: it first updates *SQTail.node->next* to point to a node consisting of the new item, and then updates *SQTail* to point to this node. An obstructing operation might enqueue its items concurrently, causing the first `CAS` (in Line 5) to fail. In this case, *EnqueueToShared* would try to help complete the obstructing operation, before starting a new attempt to enqueue its own item. This assistance is performed in Lines 9-13. Herein lies the distinction between *EnqueueToShared* in BQ and *Enqueue* in MSQ: In MSQ, the first `CAS` might fail only due to an obstructing enqueue operation, and thus only the equivalent to Line 13 of BQ is executed. In BQ, on the other hand, the obstructing operation may be either a standard enqueue operation or a batch operation.

Listing 1. EnqueueToShared

```
1 EnqueueToShared(item)
2    newNode = new Node(item, NULL)
3    while (true)
4       tailAndCnt = SQTail
```

```
5       if (CAS(&tailAndCnt.node->next, NULL, newNode))
6          // newNode is linked to the tail
7          CAS(&SQTail, tailAndCnt, ⟨newNode, tailAndCnt.cnt + 1⟩)
8          break
9       head = SQHead
10      if head consists of Ann: // (head.tag & 1 != 0)
11         ExecuteAnn(head.ann)
12      else
13         CAS(&SQTail, tailAndCnt, ⟨tailAndCnt.node->next, tailAndCnt.cnt + 1⟩)
```

DequeueFromShared. If the queue is not empty when the dequeue operation takes effect, *DequeueFromShared* extracts an item from the head of the shared queue and returns it; otherwise it returns NULL. The only addition to the MSQ's *Dequeue* is helping pending batch operations complete first by calling the *HelpAnnAndGetHead* method.

Listing 2. DequeueFromShared

```
14 Item* DequeueFromShared()
15    while (true)
16       headAndCnt = HelpAnnAndGetHead()
17       headNextNode = headAndCnt.node->next
18       if (headNextNode == NULL)
19          return NULL
20       if (CAS(&SQHead, headAndCnt, ⟨headNextNode, headAndCnt.cnt + 1⟩))
21          return headNextNode->item
```

HelpAnnAndGetHead. This auxiliary method assists announcements in execution, as long as there is an announcement installed in *SQHead*.

Listing 3. HelpAnnAndGetHead

```
22 PtrCnt HelpAnnAndGetHead()
23    while (true)
24       head = SQHead
25       if head consists of PtrCnt: // (head.tag & 1 == 0)
26          return head.ptrCnt
27       ExecuteAnn(head.ann)
```

ExecuteBatch. ExecuteBatch is responsible for executing the batch. Before it starts doing so, it checks whether there is a colliding ongoing batch operation whose announcement is installed in *SQHead*. If so, *ExecuteBatch* helps it complete (Line 31). Afterwards, it stores the current head in *ann* (Line 32), installs *ann* in *SQHead* (Line 33) and calls *ExecuteAnn* to carry out the batch. The batch execution's steps are illustrated in Figure 1.

Listing 4. ExecuteBatch

```
28 Node* ExecuteBatch(batchRequest)
29    ann = new Ann(batchRequest)
30    while (true)
31       oldHeadAndCnt = HelpAnnAndGetHead()
32       ann->oldHead = oldHeadAndCnt // Step 1 in Figure 1
33       if (CAS(&SQHead, oldHeadAndCnt, ann)) // Step 2 in Figure 1
34          break
```

```
35    ExecuteAnn(ann)
36    return oldHeadAndCnt.node
```

ExecuteAnn. *ExecuteAnn* is called with *ann* after *ann* has been installed in *SQHead*. *ann*'s *oldHead* field consists of the value of *SQHead* right before *ann*'s installation. *ExecuteAnn* carries out *anns*'s batch. If any of the execution steps has already been executed by another thread, *ExecuteAnn* moves on to the next step. Specifically, if *ann* will have been removed from *SQHead* by the time *ExecuteAnn* is executed, *ann*'s execution will have been completed, and all the steps of this run of *ExecuteAnn* would fail and have no effect.

ExecuteAnn first makes sure that *ann*'s enqueued items are linked to the queue, in the while loop in Line 39. If they have already been linked to the queue, and the old tail after which they were linked has also been recorded in *ann*, it follows that another thread has completed the linking, and thus we break out of the loop in Line 43. Otherwise, we try to link the items by performing a `CAS` operation on the next pointer of the node pointed to by the tail in Line 44. In Line 45 we check whether the items were linked after *tail*, regardless of which thread linked them. If so, we record *tail*, to which the items were linked, in *ann*. Otherwise, we try to help the obstructing enqueue operation complete in Line 50, and start over with a new attempt to link the batch's items.

The next step is *SQTail*'s update in Line 52. There is no need to retry it, since it fails only if another thread has written the same value on behalf of the same batch operation. Lastly, we call *UpdateHead* to update *SQHead* to point to the last node dequeued by the batch. This update uninstalls the announcement and completes its handling.

The *UpdateHead* method calculates *successfulDeqsNum* as described in Corollary 5.5. It then determines the new head according to the following optimization: If the number of the batch's

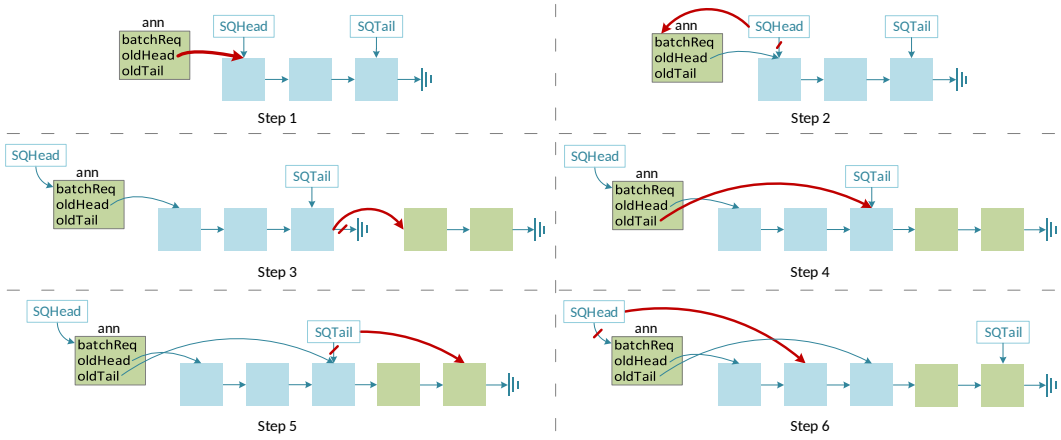

Fig. 1. Steps of the Batch Execution

(1) Setting *ann->oldHead* to the head of the queue right before the batch.
(2) Installing *ann* in *SQHead*.
(3) Linking the batch's items to *SQTail.node->next*.
(4) Setting *oldTail* field in the installed announcement *ann*.
(5) Advancing *SQTail* to point to the last node enqueued by the batch operation (and increasing its enqueue count by the number of enqueues).
(6) Setting *SQHead* to point to the last node dequeued by the batch operation in place of *ann* (and increasing its dequeue count by the number of successful dequeues).

successful dequeues is at least the size of the queue before applying the batch, which implies that the new dummy node is one of the batch's enqueued nodes, the new head is determined by passing over $successfulDeqsNum - oldQueueSize$ nodes, starting with the node pointed to by the old tail. Otherwise, it is determined by passing over $successfulDeqsNum$ nodes, starting with the old dummy node. Finally, *UpdateHead* updates *SQHead* (and as in *SQTail*'s update, there is no need to retry the CAS).

Listing 5. ExecuteAnn

```
37 ExecuteAnn(ann)
38    // Link items to tail and update ann
39    while (true)
40       tailAndCnt = SQTail
41       annOldTailAndCnt = ann->oldTail
42       if (annOldTailAndCnt.node != NULL)
43          break
44       CAS(&tailAndCnt.node->next, NULL, ann->batchReq.firstEnq) // Step 3 in Figure 1
45       if (tailAndCnt.node->next == ann->batchReq.firstEnq)
46          // Step 4 in Figure 1:
47          ann->oldTail = annOldTailAndCnt = tailAndCnt
48          break
49       else
50          CAS(&SQTail, tailAndCnt, ⟨tailAndCnt.node->next, tailAndCnt.cnt + 1⟩)
51    newTailAndCnt = ⟨ann->batchReq.lastEnq, annOldTailAndCnt.cnt + ann->batchReq.enqsNum
          ⟩
52    CAS(&SQTail, annOldTailAndCnt, newTailAndCnt) // Step 5 in Figure 1
53    UpdateHead(ann)
54
55 UpdateHead(ann)
56    oldQueueSize = ann->oldTail.cnt - ann->oldHead.cnt
57    successfulDeqsNum = ann->batchReq.deqsNum
58    if (ann->batchReq.excessDeqsNum > oldQueueSize)
59       successfulDeqsNum -= ann->batchReq.excessDeqsNum - oldQueueSize
60    if (successfulDeqsNum == 0)
61       CAS(&SQHead, ann, ann->oldHead) // Step 6 in Figure 1
62       return
63    if (oldQueueSize > successfulDeqsNum)
64       newHeadNode = GetNthNode(ann->oldHead.node, successfulDeqsNum)
65    else
66       newHeadNode = GetNthNode(ann->oldTail.node, successfulDeqsNum - oldQueueSize)
67    CAS(&SQHead, ann, ⟨newHeadNode, ann->oldHead.cnt + successfulDeqsNum⟩) // Step 6 in
          Figure 1
68
69 Node* GetNthNode(node, n)
70    repeat n times:
71       node = node->next
72    return node
```

6.2.2 Interface Methods. The queue's interface methods exposed to the user are *Enqueue*, *Dequeue*, *FutureEnqueue*, *FutureDequeue* and *Evaluate*. These methods wrap the methods that access the shared queue, which are detailed in Subsection 6.2.1. After describing them, we will elaborate on

the *PairFuturesWithResults* auxiliary method, which is called by *Evaluate*, and locally sets the futures' results to complete the batch operation.

Enqueue. *Enqueue* checks whether the thread-local operation queue *opsQueue* is empty. If it is, it directly calls *EnqueueToShared*. Otherwise, to satisfy EMF-linearizability, the pending operations in *opsQueue* must be applied before the current *Enqueue* is applied. Hence, *Enqueue* calls *FutureEnqueue* with the required item, which in turn returns a future. It then calls *Evaluate* with that future. This results in applying all preceding pending operations, as well as applying the current operation.

Listing 6. Enqueue

```
73 Enqueue(item)
74    if (threadData.opsQueue.Empty())
75       EnqueueToShared(item)
76    else
77       Evaluate(FutureEnqueue(item))
```

Dequeue. The implementation of *Dequeue* is similar to the one of *Enqueue*. If *Dequeue* succeeds, it returns the dequeued item, otherwise (the queue is empty when the operation takes effect) it returns `NULL`.

Listing 7. Dequeue

```
78 Item* Dequeue()
79    if (threadData.opsQueue.Empty())
80       return DequeueFromShared()
81    else
82       return Evaluate(FutureDequeue())
```

FutureEnqueue. *FutureEnqueue* adds the item to be enqueued to thread's list of items pending to be enqueued. This list will be appended directly to the end of the shared queue's list of nodes when a batch operation is executed by this thread. This is the reason why these items are stored in a linked list of nodes rather than directly in *opsQueue*. *FutureEnqueue* also updates the local numbers of pending enqueue operations. In addition, *FutureEnqueue* enqueues a *FutureOp* object representing an enqueue operation to the thread's *opsQueue*. A pointer to the *Future* object encapsulated in the created *FutureOp* will be returned by the method, so that the caller could later pass it to the *Evaluate* method.

Listing 8. FutureEnqueue

```
83 Future* FutureEnqueue(item)
84    AddToEnqsList(item)
85    ++threadData.enqsNum
86    return RecordOpAndGetFuture(ENQ)
87
88 AddToEnqsList(item)
89    node = new Node(item, NULL)
90    if (threadData.enqsHead == NULL)
91       threadData.enqsHead = node
92    else
93       threadData.enqsTail->next = node
94    threadData.enqsTail = node
```

```
95
96 Future* RecordOpAndGetFuture(futureOpType)
97    future = new Future()
98    threadData.opsQueue.Enqueue(⟨futureOpType, future⟩)
99    return future
```

FutureDequeue. *FutureDequeue* updates the local numbers of pending dequeue operations and excess dequeues. The latter is calculated based on Lemma 5.3. *FutureDequeue* then enqueues a *FutureOp* object representing a dequeue operation to the thread's *opsQueue*. Like *FutureEnqueue*, *FutureDequeue* returns a pointer to a *Future* object.

Listing 9. FutureDequeue

```
100 Future* FutureDequeue()
101    ++threadData.deqsNum
102    threadData.excessDeqsNum = max(threadData.excessDeqsNum, threadData.deqsNum -
           threadData.enqsNum)
103    return RecordOpAndGetFuture(DEQ)
```

Evaluate. *Evaluate* receives a future and ensures it is applied when the method returns. Notice that a future may be evaluated by its creator thread only.

If the future has already been applied from the outset, its result is immediately returned. Otherwise, all locally-pending operations found in *threadData.opsQueue* are applied to the shared queue at once. After the batch operation's execution completes, while new operations may be applied to the shared queue by other threads, the batch operation results are paired to the appropriate futures of operations in *opsQueue*.

If *opsQueue* consists of at least one enqueue operation, the batch operation's execution and the results-to-futures pairing are accomplished by calling *ExecuteBatch* and *PairFuturesWithResults* (described next) respectively. If all pending operations are dequeues, we pursue a different course of action, on which we elaborate in Subsection 6.2.3.

Listing 10. Evaluate

```
104 Item* Evaluate(future)
105    if (!future->isDone)
106       ExecuteAllPending()
107    return future->result
108
109 ExecuteAllPending()
110    if (threadData.enqsNum == 0)
111       // No enqueues. Execute a dequeues-only batch
112       ⟨successDeqsNum, oldHeadNode⟩ = ExecuteDeqsBatch()
113       PairDeqFuturesWithResults(oldHeadNode, successDeqNum)
114    else
115       // Execute a batch operation with at least one enq
116       oldHeadNode = ExecuteBatch(
117          ⟨threadData.enqsHead,
118          threadData.enqsTail,
119          threadData.enqsNum,
120          threadData.deqsNum,
121          threadData.excessDeqsNum⟩)
122       PairFuturesWithResults(oldHeadNode)
```

```
123       threadData.enqsHead = NULL
124       threadData.enqsTail = NULL
125       threadData.enqsNum = 0
126    threadData.deqsNum = 0
127    threadData.excessDeqsNum = 0
```

PairFuturesWithResults. PairFuturesWithResults receives the old head. It simulates the pending operations one by one according to their original order, which is recorded in the thread's *opsQueue*. Namely, it simulates updates of the head and tail of the shared queue. This is done by advancing *nextEnqNode* (which represents the value of *tail->next* in the current moment of the simulation) on each enqueue, and by advancing *currentHead* on dequeues that occur when the queue in its current state is not empty. The simulation is run in order to set results for future objects related to the pending operations and mark them as done.

Listing 11. PairFuturesWithResults

```
128 PairFuturesWithResults(oldHeadNode)
129    nextEnqNode = threadData.enqsHead
130    currentHead = oldHeadNode
131    noMoreSuccessfulDeqs = false
132    while threadData.opsQueue is not empty:
133       op = threadData.opsQueue.Dequeue()
134       if (op.type == ENQ)
135          nextEnqNode = nextEnqNode->next
136       else // op.type == DEQ
137          if (noMoreSuccessfulDeqs ||
138             currentHead->next == nextEnqNode)
139             // The queue is currently empty
140             op.future->result = NULL
141          else
142             currentHead = currentHead->next
143             if (currentHead == threadData.enqsTail)
144                noMoreSuccessfulDeqs = true
145             op.future->result = currentHead->item
146       op.future->isDone = true
```

6.2.3 Dequeues-Only Batch. The batch execution scheme outlined in Subsection 5.1 and detailed in Subsection 6.2.1 does not work if the batch operation consists solely of dequeue operations. In such scenario, the *Evaluate* method calls *ExecuteDeqsBatch* to apply the batch operation. The *ExecuteDeqsBatch* method first assists a colliding ongoing batch operation if there is any (in Line 149). It then calculates the new head and the number of successful dequeues by traversing over the items to be dequeued in the loop in Line 152. If there is at least one successful dequeue, the dequeues take effect at once using a single CAS operation in Line 160. The CAS pushes the shared queue's head $successfulDeqsNum$ nodes forward.

Then *Evaluate* calls *PairDeqFuturesWithResults* to pair the successfully-dequeued-items to futures of the appropriate operations in *opsQueue*. The remaining future dequeues are unsuccessful, thus their results are set to NULL.

Listing 12. ExecuteDeqsBatch

```
147 ⟨unsigned int, Node*⟩ ExecuteDeqsBatch()
148    while (true)
```

```
149       oldHeadAndCnt = HelpAnnAndGetHead()
150       newHeadNode = oldHeadAndCnt.node
151       successfulDeqsNum = 0
152       repeat threadData.deqsNum times:
153          headNextNode = newHeadNode->next
154          if (headNextNode == NULL)
155             break
156          ++successfulDeqsNum
157          newHeadNode = headNextNode
158       if (successfulDeqsNum == 0)
159          break
160       if (CAS(&SQHead, oldHeadAndCnt, ⟨newHeadNode, oldHeadAndCnt.cnt +
              successfulDeqsNum⟩))
161          break
162    return ⟨successfulDeqsNum, oldHeadAndCnt.node⟩
```

Listing 13. PairDeqFuturesWithResults

```
163 PairDeqFuturesWithResults(oldHeadNode, successfulDeqsNum)
164    currentHead = oldHeadNode
165    repeat successfulDeqsNum times:
166       currentHead = currentHead->next
167       op = threadData.opsQueue.Dequeue()
168       op.future->result = currentHead->item
169       op.future->isDone = true
170    repeat threadData.deqsNum - successfulDeqsNum times:
171       op = threadData.opsQueue.Dequeue()
172       op.future->result = NULL
173       op.future->isDone = true
```

7 MEMORY MANAGEMENT

We utilized the optimistic access scheme [2], which extends the hazard pointers scheme [21], as a lock-free manual memory management mechanism for BQ. All measurements include use of memory reclamation.

We describe memory management of lock-free data structures in general in Subsection 7.1, and explain the optimistic access mechanism. Then, we describe how this mechanism is utilized in BQ in Subsection 7.2.

7.1 Lock-Free Manual Memory Management

A lock-free data structure requires a delicate memory reclamation mechanism. Such mechanism should prevent two risks posed by reclamation: an access to shared memory that has been freed by another thread, and the ABA problem (comparing a pointer to an expected value that has been recycled). No efficient lock-free automatic garbage collector exists in literature. Therefore, to manage memory of lock-free data structures in a lock-free fashion, one should employ a manual memory management scheme. In such schemes, an object that is part of the shared data structure is reclaimed in coordination between the data structure's algorithm and the reclamation procedure: First, the algorithm unlinks the object from the data structure. Next, to declare that the object is no longer needed, the algorithm announces it as retired. This implies that the object should be reclaimed when with certainty no one might access it or compare its address anymore. A reclamation procedure runs periodically or when there is not enough free memory space, and

reclaims the nodes that were announced as retired so far *and* are guaranteed not be accessed or compared later. Each manual memory management scheme dictates a different approach to determine which retired nodes are safe to reclaim.

The lock-free memory management scheme we utilize in our measured implementations of BQ, MSQ and Kogan and Herlihy's queue is optimistic access [2]. It employs hazard pointers [21] for write operations. In the hazard pointers method, each thread owns single-writer multi-reader shared pointers called hazard pointers. A thread assigns hazard pointers indicating memory locations it might later access or compare, in order to protect them from reclamation. This scheme considers a retired node as safe to reclaim if no hazard pointer points to it.

Read operations are performed in the optimistic access method without installing hazard pointers and setting memory fences: A read operation first reads the data without a prior check, and only then verifies that the read memory location has not been reclaimed. The verification will mostly succeed, but when it does not, the operation should be restarted as it might have read reclaimed memory. To enable reading a possibly deallocated address without triggering a segmentation fault, the algorithm utilizes a user-level allocator. This allocator maintains a pool of objects and does not return pages back to the operating system.

A verification is carried out in both read and write operations: in read operations, after reading a value we verify its memory has not been reclaimed; in write operations, after setting a hazard pointer to an address we verify this address is still safe to use. The verification in optimistic access is performed by confirming that a thread-local flag is not set: To signal that a reclamation phase has started and every object retired so far might be recycled, optimistic access maintains a local warning flag per thread. This flag is set during the reclamation process.

7.2 Applying the Optimistic Access Scheme to BQ

We adapted the optimistic access scheme presented in [2] to our needs. To begin with, we extended it to support both requisite objects (*Node* and *Ann*). For additional details about the adjustment of the original optimistic access mechanism to BQ, refer to Subsection 7.2.1.

When applying memory management to BQ, we had to make sure that a dequeued item is read before it is retired, so that it could be returned to its dequeuer. This requires some delicate manipulations, described in Subsection 7.2.2 under Retirements.

We also made some optimizations upon the conservative optimistic access usage scheme. In writes to shared locations, we assign hazard pointers only to relevant addresses that might be retired and not to all related addressed. Moreover, we do not use a `CAS` where it is not necessary in contrast to the specification of the original scheme. We further explain about write optimizations in Subsection 7.2.2 under Writes.

We apply another optimization when finding out, during a batch operation's execution, that a warning flag is set. In such cases we refrain from starting the execution from scratch, and instead perform an additional check to determine if we may proceed. See details in Subsection 7.2.2 in the part that discusses avoiding batch execution restart.

Next we elaborate further on the adjustment of the optimistic access mechanism to BQ and its usage throughout BQ's algorithm.

7.2.1 Adjusting the Optimistic Access Mechanism. We utilize the optimistic access mechanism described in Section 4 in [2], and extend it to handle two object sizes - *Node*'s size and *Ann*'s size. We call $MM.Retire$ to trigger the mechanism's *Reclaim* function, and $MM.AllocateNode$ and $MM.AllocateAnn$ to trigger its *Allocate* function for the appropriate object size.

BQ's $ThreadData$ entry held by each thread is extended to include the memory management related data: a *warning* Boolean flag as well as hazard pointers (*nodeHp* and *annHp*).

7.2.2 Adjusting BQ's Code. We cover the necessary modifications that must be applied to allocations, retirements, writes and reads from the shared memory in BQ. The code modifications are presented thereafter.

Allocations. Objects are allocated by a user-level allocator. It allocates a node for a standard enqueue operation in Line 175, and for a future enqueue operation in Line 377. An announcement object is allocated in Line 237.

Retirements. An announcement is retired in Line 253 by the same method that created it. Nodes could be retired in several occasions, depending on the way they are dequeued: A node dequeued by a single dequeue operation is retired in Line 219 right after advancing the queue's head to the next node. A node dequeued by a batch operation is retired when traversing it during the pairing process of the batch's applied future operations with results (in Line 435 in case of a batch that includes both enqueues and dequeues; or in Lines 453 and 457 for nodes dequeued by a dequeues-only batch operation).

Retiring nodes should be done carefully to ensure retrieving a dequeued item prior to the recycling of its enclosing node. Next, we describe the difficulties in achieving this goal and then detail how we accomplish it.

Recall that the queue's head points to a dummy node. Consequently, each node's matching item lies in its successor node. Let A be an address of a node pointed to by the head, and let *newHead* equal A->*next*, which is a pointer to A's successor. A is dequeued by setting the head to *newHead*, retiring A and returning *newHead->item* as the dequeued item. This dequeue operation, which returns *newHead->item*, is not the same one that retires *newHead*. We should make sure to read *newHead->item* when *newHead* is still certainly not retired. Otherwise (i.e., if another dequeue or batch operation retire *newHead* beforehand), then the node pointed to by *newHead* may be recycled, in which case the dequeuing thread might not be able to get a hold of its dequeued item.

In a single dequeue operation, we make sure to read the item prior to its recycling by reading *newHead->item* before applying a `CAS` to the queue's head. If we read *newHead->item* and then successfully `CAS` the head, we know for certain that when we read *newHead->item*, *newHead* has not yet been retired: a node is retired only during the execution of its dequeuing by a dequeue or a future dequeue operation, and the node pointed to by *newHead* could not have been dequeued before the `CAS` of head from A to *newHead* succeeded.

Accomplishing the goal of retrieving an item dequeued by a future operation prior to its recycling is more tricky. The reason is that during a batch execution, we do not read all dequeued items before applying a `CAS` to the queues' head, because we aim to minimize the synchronization time. Thus, only after `CAS`ing the head to complete the batch's effect on the shared queue, does the thread that initiated the batch pair its applied futures with results locally. During the pairing process, the initiating thread traverses its successfully-dequeued nodes, reads their corresponding items and retires them. Reading all dequeued items but the last one can be easily performed before retiring the nodes that contain them, as the initiating thread is the one responsible for their retirement. We read the items' values in Lines 443 and 455 of PairFuturesWithResults and PairDeqFuturesWithResults methods respectively, before retiring the node that holds them in the next loop iteration in Line 435 of PairFuturesWithResults and Lines 453 and 457 of PairDeqFuturesWithResults.

The last dequeued item is problematic: It lies in the node that is pointed by the head after the batch execution. This node is retired by the thread that performs the subsequent dequeue or batch operation. This might not be our batch's initiator, but rather another thread.

Therefore, like in a single dequeue operation, the thread that executes the batch operation should read the last dequeued item before performing a `CAS` of the head, since after this `CAS` occurs, the node that holds the last dequeued item might be recycled. The remaining question is how this read

value is later paired with the appropriate future. The answer depends on the kind of batch that the future dequeue is a part of.

Let us examine a dequeues-only batch first. It requires a single modification to the shared queue's state: a CAS of the head. Therefore, no helping is involved in its execution. The batch's initiator is the only one to apply it to the shared queue, and then match the futures with the results. It reads the last dequeued item, which lies in the node that is about to be pointed to by the new head, in Line 279. Just like in a single dequeue operation, the item is read before applying a CAS to the queue's head, which enables a recycling of the node that holds this item. Then, as the initiating thread is also the one to pair the batch's futures with results, it simply sets the future's result of the last successful dequeue to this read value in Line 459.

The case of a batch operation that contains both dequeues and enqueues is more complicated, because the thread that initiates such batch operation is the one responsible for pairing its futures with results, but the batch execution itself may be carried out by a helping thread. The thread that executes the batch operation should read the last dequeued item before performing the CAS that uninstalls the announcement from the head, and inform the initiating thread of this item. This is mandatory since after the CAS of the head, by the time the future of the last successful dequeue is paired with the appropriate item, the node that holds this item might have already been recycled. To inform the initiating thread about the read item, the thread that executes the batch should keep it in some shared location. We chose to place it in the node that was pointed to by the queue's head prior to the current announcement's installation. This node is under the responsibility of the batch's initiator who should retire it, thus no one else relies on its content (due to the same arguments). The last dequeued item is read in Line 344 and placed in the node pointed to by the old head in Line 348 (after verifying that the item was read before its node was possibly changed by a subsequent batch operation or retired). Later, the future of the last successful dequeue is paired with this item in Line 446 of Listing 25.

The batch's applied futures are paired with results by the thread that initiated the batch (in Method *PairFuturesWithResults* in Listing 25). However, this thread has not necessarily carried out the batch on its own, so it might not know which item is the last to be successfully dequeued from the queue. Therefore, when it traverses over the batch's operations and pairs them with results, it does not know if a current successful dequeue is the last successful one and its item was stored in the node pointed to by the old head, or it is not the last and its item lies in the next node. To circumvent this problem, on each successful dequeue the initiator encounters, it sets the result of the previous successful dequeue, which is now revealed not to be the last. When the traversal is over, the last successful future dequeue is revealed, and its result is set to the item that was stored in the node pointed to by the old head node in advance.

Writes. Before each write to a shared location, we set a hazard pointer to the hazardous reference, apply a memory fence to ensure the hazard pointer is visible to all threads, and check the warning flag to verify we may proceed.

The original optimistic access paper described a conservative approach to protect writes to shared locations: writing only using a CAS, and setting hazard pointers to all associated pointers (the location that is about to be written, its expected value and the new value). However, in most writes, optimizations may apply, and so we avoid any of these hazard pointers where they are not required, as well as refrain from using an avoidable CAS.

One kind of writes we guard is to long-lasting shared locations: the shared queue's current head and tail pointers. This kind of write carries the risk of the ABA problem, in case the CAS's expected value is recycled. Hence we install a hazard pointer to the expected pointer value of head before performing a CAS of the shared queue's head (in Lines 207, 241 and 258, where head's previous

value is a pointer to a node, and in Lines 192 and 228, where its previous value is a pointer to an announcement) and similarly for tail (in Lines 179, 293 and 301). The location to which we write is not subjected to an access hazard, since it is not freed during the whole run.

On the other hand, an access to a short-lasting shared object (a node or an announcement) is subjected to an access hazard. For this type of writes, we install a hazard pointer to the target object's address prior to the access, to prevent recycling of the target object. We guard a tail pointer before the next field of the node it points to is updated when linking new nodes (in Lines 179 and 301). A node to which we wish to write a batch's last dequeued item is guarded first (in Line 330). Likewise, an announcement object is protected before its batch execution, which involves setting its *oldTail* field (in Lines 192 and 228).

The two kinds of writes we described sometimes overlap. In such cases, an installation of a certain hazard pointer serves as a safeguard against both potential problems.

Reads. Reads are treated differently: After reading a node's next pointer, we set a compiler fence to prevent compiler reordering. We then check the warning flag to verify the node has not been recycled, and thus the address we read is valid. This occurs when traversing over the nodes to be dequeued for finding the new head after a batch operation (in Lines 269 and 356).

Avoid batch execution restart. We apply the following optimization throughout the execution of a batch in Method *ExecuteAnn* (Listing 19): If we test the warning flag and find it set, then according to the optimistic access scheme guidelines we should unset it and restart the operation. Restarting in the current stage of execution would mean to reread the queue's head value, and if it contains an announcement, start its execution from the beginning.

Instead of restarting, when discovering that our warning flag is set, we check whether the announcement we hold is still installed in the queue's head. If not, the batch execution has anyhow been completed by another thread, so we may immediately return. Otherwise, we may proceed with the batch execution: When we find the warning set in Line 356 after reading *node->next*, the still-installed announcement implies that when we read the next pointer it has not been retired, because it is located in a later node in the queue that cannot be retired as long as the announcement is installed. Similarly, we might find the warning set after assigning a hazard pointer to the node that was pointed to by the head or tail prior to the batch execution. In such case, the still-installed announcement implies that when we set the hazard pointer, the node pointed to by the old head or tail has not been retired.

The modified code. The additions to the algorithm presented in Section 6 are colored in red in the following code snippets.

We begin with the implementation of the internal methods operating on the shared queue.

Listing 14. EnqueueToShared

```
174 EnqueueToShared(item)
175    node = MM.AllocateNode()
176    node.item = item; node.next = NULL
177    while (true)
178       tailAndCnt = SQTail
179       threadData.nodeHp = tailAndCnt.node
180       _memoryFence
181       if (threadData.warning)
182          threadData.warning = false
183          threadData.nodeHp = NULL
184          continue
```

```
185       if (CAS(&tailAndCnt.node->next, NULL, node))
186          // Linked node to tail
187          CAS(&SQTail, tailAndCnt, ⟨node, tailAndCnt.cnt+1⟩)
188          break
189       head = SQHead
190       if head consists of Ann: // (head.tag & 1 != 0)
191          threadData.nodeHp = NULL
192          threadData.annHp = head.ann
193          _memoryFence
194          if (threadData.warning)
195             threadData.warning = false
196             threadData.annHp = NULL
197             continue
198          ExecuteAnn(head.ann)
199          threadData.annHp = NULL
200       else
201          CAS(&SQTail, tailAndCnt, ⟨tailAndCnt.node->next, tailAndCnt.cnt+1⟩)
202          threadData.nodeHp = NULL
203    threadData.nodeHp = NULL
```

Listing 15. DequeueFromShared

```
204 Item* DequeueFromShared()
205    while (true)
206       headAndCnt = HelpAnnAndGetHead()
207       threadData.nodeHp = headAndCnt.node
208       _memoryFence
209       if (threadData.warning)
210          threadData.warning = false
211          goto cleanup
212       headNextNode = headAndCnt.node->next
213       if (headNextNode == NULL)
214          threadData.nodeHp = NULL
215          return NULL
216       dequeuedItem = headNextNode->item
217       if (CAS(&SQHead, headAndCnt, ⟨headNextNode, headAndCnt.cnt+1⟩))
218          threadData.nodeHp = NULL
219          MM.Retire(headAndCnt.node)
220          return dequeuedItem
221       cleanup:
222       threadData.nodeHp = NULL
```

Listing 16. HelpAnnAndGetHead

```
223 PtrCnt HelpAnnAndGetHead()
224    while (true)
225       head = SQHead
226       if head consists of PtrCnt: // (head.tag & 1 == 0)
227          return head.ptrCnt
228       threadData.annHp = head.ann
229       _memoryFence
230       if (threadData.warning)
231          threadData.warning = false
```

```
232          threadData.annHp = NULL
233          continue
234       ExecuteAnn(head.ann)
235       threadData.annHp = NULL
```

Listing 17. ExecuteBatch

```
236 Node* ExecuteBatch(batchRequest)
237    ann = MM.AllocateAnn()
238    ann->batchReq = batchRequest
239    while (true)
240       oldHeadAndCnt = HelpAnnAndGetHead()
241       threadData.nodeHp = oldHeadAndCnt.node
242       _memoryFence
243       if (threadData.warning)
244          threadData.warning = false
245          goto cleanup
246       ann->oldHead = oldHeadAndCnt // Step 1 in Figure 1
247       if (CAS(&SQHead, oldHeadAndCnt, ann)) // Step 2 in Figure 1
248          break
249       cleanup:
250       threadData.nodeHp = NULL
251    threadData.nodeHp = NULL
252    ExecuteAnn(ann)
253    MM.Retire(ann)
254    return oldHeadAndCnt.node
```

Listing 18. ExecuteDeqsBatch

```
255 ⟨unsigned int, Node*, Item*⟩ ExecuteDeqsBatch()
256    while (true)
257       oldHeadAndCnt = HelpAnnAndGetHead()
258       threadData.nodeHp = oldHeadAndCnt.node
259       _memoryFence
260       if (threadData.warning)
261          threadData.warning = false
262          goto cleanup
263       newHeadNode = oldHeadAndCnt.node
264       successfulDeqsNum = 0
265       // Calculate new head and successful dequeues num:
266       repeat threadData.deqsNum times:
267          headNextNode = newHeadNode->next
268          _compilerFence
269          if (threadData.warning)
270             threadData.warning = false
271             goto cleanup
272          if (headNextNode == NULL)
273             break
274          ++successfulDeqsNum
275          newHeadNode = headNextNode
276       if (successfulDeqsNum == 0)
277          lastDeqItem = NULL
278          break
```

```
279       lastDeqItem = newHeadNode->item
280       if (CAS(&SQHead, oldHeadAndCnt, ⟨newHeadNode, oldHeadAndCnt.cnt +
              successfulDeqsNum⟩))
281          break
282       cleanup:
283       threadData.nodeHp = NULL
284    threadData.nodeHp = NULL
285    return ⟨successfulDeqsNum, oldHeadAndCnt.node, lastDeqItem⟩
```

Listing 19. ExecuteAnn

```
286 // ann is assured not to be reclaimed during ExecuteAnn run
287 ExecuteAnn(ann)
288    // Link items to tail and update ann
289    while (true)
290       tailAndCnt = SQTail
291       annOldTailAndCnt = ann->oldTail
292       if (annOldTailAndCnt.node != NULL)
293          threadData.nodeHp = annOldTailAndCnt.node
294          _memoryFence
295          if (threadData.warning)
296             threadData.warning = false
297             if (SQHead != ann)
298                threadData.nodeHp = NULL
299                return
300          break
301       threadData.nodeHp = tailAndCnt.node
302       _memoryFence
303       if (threadData.warning)
304          threadData.warning = false
305          if (SQHead != ann)
306             threadData.nodeHp = NULL
307             return
308       CAS(&tailAndCnt.node->next, NULL, ann->batchReq.firstEnq) // Step 3 in Figure 1
309       if (tailAndCnt.node->next == ann->batchReq.firstEnq)
310          // Step 4 in Figure 1:
311          ann->oldTail = annOldTailAndCnt = tailAndCnt
312          _memoryFence
313          break
314       else
315          CAS(&SQTail, tailAndCnt, ⟨tailAndCnt.node->next, tailAndCnt.cnt+1⟩)
316       threadData.nodeHp = NULL
317    newTailAndCnt = ⟨ann->batchReq.lastEnq, annOldTailAndCnt.cnt + ann->batchReq.enqsNum
          ⟩
318    CAS(&SQTail, annOldTailAndCnt, newTailAndCnt) // Step 5 in Figure 1
319    threadData.nodeHp = NULL
320    UpdateHead(ann)
321
322 UpdateHead(ann)
323    oldQueueSize = ann->oldTail.cnt - ann->oldHead.cnt
324    successfulDeqsNum = ann->batchReq.deqsNum
325    if (ann->batchReq.excessDeqsNum > oldQueueSize)
```

```
326       successfulDeqsNum -= ann->batchReq.excessDeqsNum - oldQueueSize
327    if (successfulDeqsNum == 0)
328       CAS(&SQHead, ann, ann->oldHead) // Step 6 in Figure 1
329       return
330    threadData.nodeHp = ann->oldHead.node
331    _memoryFence
332    if (threadData.warning)
333       threadData.warning = false
334       if (SQHead != ann)
335          threadData.nodeHp = NULL
336          return
337    if (oldQueueSize > successfulDeqsNum)
338       newHeadNode = GetNthNode(ann->oldHead.node, successfulDeqsNum, ann)
339    else
340       newHeadNode = GetNthNode(ann->oldTail.node, successfulDeqsNum - oldQueueSize, ann
           )
341    if (newHeadNode == NULL)
342       threadData.nodeHp = NULL
343       return
344    lastDequeuedItem = newHeadNode->item
345    if (SQHead != ann)
346       threadData.nodeHp = NULL
347       return
348    ann->oldHead.node->item = lastDequeuedItem
349    threadData.nodeHp = NULL
350    CAS(&SQHead, ann, ⟨newHeadNode, ann->oldHead.cnt + successfulDeqsNum⟩) // Step 6 in
        Figure 1
351
352 Node* GetNthNode(node, n, ann)
353    repeat n times:
354       node = node->next
355       _compilerFence
356       if (threadData.warning)
357          threadData.warning = false
358          if (SQHead != ann)
359             return NULL
360    return node
```

Next we present the implementation of the queue's interface methods.

Listing 20. Enqueue

```
361 Enqueue(item)
362    if (threadData.opsQueue.Empty())
363       EnqueueToShared(item)
364    else
365       Evaluate(FutureEnqueue(item))
```

Listing 21. Dequeue

```
366 Item* Dequeue()
367    if (threadData.opsQueue.Empty())
368       return DequeueFromShared()
```

```
   else
      return Evaluate(FutureDequeue())
```

Listing 22. FutureEnqueue

```
Future* FutureEnqueue(item)
   AddToEnqsList(item)
   ++threadData.enqsNum
   return RecordOpAndGetFuture(ENQ)

AddToEnqsList(item)
   node = MM.AllocateNode()
   node.item = item; node.next = NULL
   if (threadData.enqsHead == NULL)
      threadData.enqsHead = node
   else
      threadData.enqsTail->next = node
   threadData.enqsTail = node

Future* RecordOpAndGetFuture(futureOpType)
   future = new Future()
   threadData.opsQueue.Enqueue(⟨futureOpType, future⟩)
   return future
```

Listing 23. FutureDequeue

```
Future* FutureDequeue()
   ++threadData.deqsNum
   threadData.excessDeqsNum = max(threadData.excessDeqsNum, threadData.deqsNum -
        threadData.enqsNum)
   return RecordOpAndGetFuture(DEQ)
```

Listing 24. Evaluate

```
Item* Evaluate(future)
   if (!future->isDone)
      ExecuteAllPending()
   return future->result

ExecuteAllPending()
   if (threadData.enqsNum == 0)
      // No enqueues. Execute a dequeues-only batch
      ⟨successfulDeqsNum, oldHeadNode, lastDeqItem⟩ = ExecuteDeqsBatch()
      PairDeqFuturesWithResults(oldHeadNode, successfulDeqNum, lastDeqItem)
   else
      // Execute a batch operation with at least one enq
      oldHeadNode = ExecuteBatch(
         ⟨threadData.enqsHead,
         threadData.enqsTail,
         threadData.enqsNum,
         threadData.deqsNum,
         threadData.excessDeqsNum⟩)
      PairFuturesWithResults(oldHeadNode)
```

```
412        threadData.enqsHead = NULL
413        threadData.enqsTail = NULL
414        threadData.enqsNum = 0
415     threadData.deqsNum = 0
416     threadData.excessDeqsNum = 0
```

Listing 25. PairFuturesWithResults

```
417 PairFuturesWithResults(oldHeadNode)
418     nextEnqNode = threadData.enqsHead
419     currentHead = oldHeadNode
420     noMoreSuccessfulDeqs = false
421     shouldSetPrevDeqResult = false
422     lastSuccessfulDeqFuture = NULL
423     oldHeadItem = oldHeadNode->item
424     while threadData.opsQueue is not empty:
425        op = threadData.opsQueue.Dequeue()
426        if (op.type == ENQ)
427           nextEnqNode = nextEnqNode->next
428        else // op is DEQ
429           if (noMoreSuccessfulDeqs || currentHead->next == nextEnqNode)
430              // The queue is currently empty
431              op.future->result = NULL
432           else
433              nodePrecedingDeqNode = currentHead
434              currentHead = currentHead->next
435              MM.Retire(nodePrecedingDeqNode)
436              if (currentHead == threadData.enqsTail)
437                 noMoreSuccessfulDeqs = true
438              if (shouldSetPrevDeqResult)
439                 lastSuccessfulDeqFuture->result = lastSuccessfulDeqItem
440              else
441                 shouldSetPrevDeqResult = true
442              lastSuccessfulDeqFuture = op.future
443              lastSuccessfulDeqItem = currentHead->item
444        op.future->isDone = true
445     if (shouldSetPrevDeqResult)
446        lastSuccessfulDeqFuture->result = oldHeadItem
```

Listing 26. PairDeqFuturesWithResults

```
447 PairDeqFuturesWithResults(oldHeadNode, successfulDeqsNum, lastDeqItem)
448     if (successfulDeqsNum > 0)
449        currentHead = oldHeadNode
450        repeat successfulDeqsNum-1 times:
451           nodePrecedingDeqNode = currentHead
452           currentHead = currentHead->next
453           MM.Retire(nodePrecedingDeqNode)
454           op = threadData.opsQueue.Dequeue()
455           op.future->result = currentHead->item
456           op.future->isDone = true
457        MM.Retire(currentHead)
458        op = threadData.opsQueue.Dequeue()
```

```
459       op.future->result = lastDeqItem
460       op.future->isDone = true
461     repeat threadData.deqsNum-successfulDeqsNum times:
462       op = threadData.opsQueue.Dequeue()
463       op.future->result = NULL
464       op.future->isDone = true
```

8 CORRECTNESS

In this section we prove the correctness (i.e., linearizability) and progress guarantee (lock-freedom) of BQ.

8.1 Linearizability Proof

To prove that BQ is linearizable, we first define in Section 8.1.1 a translation of an execution on BQ to a linearization – a sequential execution on an abstract queue. The abstract queue is an imaginary sequential FIFO queue on which the linearization's operations are executed, and its state represents the state of the queue in the original execution. By *translation* we refer to picking a linearization point for each operation in the original execution, which is a moment during the operation's execution when it is applied to the abstract queue. From this translation, we derive in Section 8.1.2 the correspondence in every moment between the abstract queue's state in the linearization and BQ's underlying list of nodes in the original execution. This correspondence helps to reason about the state of the queue in each moment. In Section 8.1.3, we confirm that the results of operations performed in an execution on BQ are the same as the results of applying these operations in their linearization points to the abstract queue. This is what linearizability requires – that operating on BQ is equivalent to a linearization, namely, a legal execution on a sequential FIFO queue. In addition, we show in Section 8.1.4 that no operation is applied twice, in spite of concurrent assisting threads attempting to execute the same operation, by proving that for each operation, a single linearization point is reached (by any of the threads). For simplicity, we ignore the nodes' memory reclamation throughout our proof.

8.1.1 Linearization Points. We define linearization points for all operation types, starting with single (non-batched) operations. A successful dequeue operation takes effect when it modifies the head pointer (and counter). An unsuccessful dequeue operation is linearized when it reads the next pointer of the dummy node (whose value is revealed to be `NULL`). An enqueue operation takes effect when the next pointer of the last node in the underlying list is modified from `NULL` to point to a new node.

We proceed to listing the linearization points of batch operations. All enqueues and dequeues of a batch operation that enqueues at least one item take effect one after another when the next pointer of the last node is modified from `NULL` to point to the first node of the batch. Note that this is the only linearization step that may be carried out by a helping thread rather than the thread that invoked the operation.

Regarding a batch operation that contains only dequeue operations, all dequeues of such batch operation are linearized one after another, in the moment of reading the next pointer in Line 153 (of Method *ExecuteDeqsBatch* in Listing 12) for the last time before *ExecuteDeqsBatch* returns. This reading ends the list traversal performed to calculate the new head, either due to encountering the end of the list (as detected in Line 154), or due to completing a traversal of *deqsNum* items. The later advance of the head, which completes the dequeues batch, might intuitively seem like a simpler linearization point, nevertheless it is not a correct linearization point in all cases. Consider the following scenario: A thread T performs a dequeues-only batch. During the traversal over the

nodes to be dequeued, it encounters the end of the nodes list, after traversing less than $deqsNum$ items. Then, another thread enqueues an item, before T advances the head to point to the last dequeued node. In this scenario, the head modification by T cannot be considered the batch's linearization point, since it happens after the enqueue operation, while the batch operation does not dequeue the new item. On the other hand, the moment in which T read the next pointer in Line 153 for the last time occurred appropriately before the enqueue.

To satisfy EMF-linearizability, the future history H_f, constructed from the original history H (as described in Definition 3.1), should be MF-linearizable. Thus far, we defined linearization points relating to H. We set the same linearization points in H_f: the linearization point in H_f is the same as in H for a future operation call, and for a single operation call – we set the linearization point of the appropriate future call to the same moment as the linearization of the single call in H. Note that since the linearization points defined for single operations occur during their method calls in H, they occur between the adequate future call's invocation and *Evaluate* call's response in H_f, which complies with MF-linearizability.

8.1.2 The Abstract State of the Queue. From the above linearization point definitions, we derive the abstract state of the queue (and in particular the abstract head), with respect to the shared queue's underlying list of nodes. The abstract state of the queue is the sequence of items contained in the underlying list's nodes, starting with the item in the second node (i.e., the node succeeding the dummy node pointed to by the abstract head) if any, and ending with the node whose next pointer is `NULL`. The queue is empty iff the next pointer of the node pointed to by the abstract head is `NULL`. The tail pointer does not affect the state of the abstract queue.

The fine point is the definition of the abstract head of the queue:

- If no announcement is installed in *SQHead*, the abstract head is the same as *SQHead*. There is one exception to this rule: a dequeues-only batch operation that succeeds to dequeue at least one item[1]. Such operation modifies the abstract head: when it reads the next pointer in Line 153 (of Method *ExecuteDeqsBatch* in Listing 12) for the last time before *ExecuteDeqsBatch* returns, the abstract head is set to point to the last node dequeued by this batch operation. The pointer to this node is the value to which *SQHead* is set in Line 160 in the same execution of *ExecuteDeqsBatch*.
- If there is an announcement installed in *SQHead*, but the `CAS` that links its items to the tail (in Line 44 in Listing 5) has not yet been performed successfully, then the abstract head is the same as *SQHead.ann->oldHead*, which is in practice *SQHead*'s value prior to the announcement's installation. Thus, installing an announcement does not change the abstract head.
- If there is an announcement installed in *SQHead*, and the `CAS` that links its items to the tail has already succeeded, then the abstract head points to the node that is going to be the dummy node after all enqueues and dequeues of the batch operation have taken effect. *SQHead* is going to be set to the same value when the announcement is uninstalled. Thus, removing an announcement does not change the abstract head.

Hence, when a batch operation is announced (i.e., *SQHead* is set to point to the related announcement), the abstract state of the queue does not change. It remains the sequence of items currently contained in the nodes of the shared queue's list, starting with the node succeeding the node pointed to by the previous *SQHead*. The moment the `CAS` that links the batch's enqueued items to the tail (in Line 44 in Listing 5) succeeds, the abstract queue's head is changed to point to the node to which *SQHead* will point after completing the announcement handling. Therefore, the

[1]In the short version of the paper we did not elaborate on the effect of a dequeues-only batch on the abstract head.

whole batch, including both its enqueues and dequeues, takes effect instantaneously. From that moment, until another operation takes effect, the abstract state of the queue is the sequence of items in the list starting with the node succeeding the new dummy node, including the batch's linked items.

8.1.3 Operation Results Comply with the Sequential Specification of a FIFO Queue. We will show that applying operations to BQ is equivalent (in terms of their results) to applying the same operations, in their linearization point moments, to the sequential FIFO abstract queue. We start with operations that affect the shared queue's state.

Regarding a non-batched dequeue operation, a successful dequeue's linearization point is in Line 20 in Listing 2, where *SQHead* is updated to point to the next node. Note that the abstract head before and after the update is the same as *SQHead*. This is because no batch operations are involved: First, no announcement is installed in the head at this moment, since both the previous and new *SQHead* values contain *PtrCnt* objects. Second, no dequeues-only batch operation has advanced the abstract head. To do so it needs to later succeed advancing the current *SQHead*, which is impossible as the present dequeue is the one to succeed performing a `CAS` of *SQHead*. Thus, the `CAS` in Line 20, which advances *SQHead* by one node, advances the abstract head as well. This translates to dequeuing the first item from the abstract queue.

As to a dequeues-only batch operation that takes effect when the queue is not empty – all its dequeues are linearized successively, in the moment of reading the next pointer in Line 153 (of Method *ExecuteDeqsBatch* in Listing 12) for the last time before *ExecuteDeqsBatch* returns. We denote this moment by t_2. Let t_1 be the moment when *SQHead*'s value is obtained in Line 149 for the last time before *ExecuteDeqsBatch* returns, and let H be the obtained value. The abstract head at t_1 is H, according to the abstract head's definition in Section 8.1.2: H does not point to an announcement (as guaranteed by *HelpAnnAndGetHead*), and since *ExecuteDeqsBatch* is about to succeed to `CAS` the head from H – no other dequeues batch has succeeded to do this. The abstract head remains H from t_1 until t_2: since our dequeues batch is about to succeed to `CAS` the head from H to point to another node, no other dequeue or batch succeeds to do this (i.e., during this time, no dequeue or another batch that contains dequeues is applied). If the last next pointer to be read in Line 153 at t_2 is `NULL`, then right before this read there are exactly $successfulDeqsNum$ items in the abstract queue (because the abstract head is still H as explained above, and we traversed $successfulDeqsNum$ nodes, linked after the abstract head, until reaching the end of the list). In this case, the abstract head is advanced at t_2 by $successfulDeqsNum$ nodes and the queue becomes empty. The rest of the dequeues in the batch fail. If, on the other hand, the last next pointer to be read in Line 153 at t_2 is not `NULL`, then right before this read there are at least $threadData.deqsNum$ items in the queue. The abstract head is advanced at t_2 by $threadData.deqsNum$ nodes, which translates to an extraction of this amount of items from the beginning of the queue. In any case, *SQHead* is eventually advanced in Line 160 of *ExecuteDeqsBatch*, in a `CAS` that does not affect the abstract state of the queue.

Moving to enqueue, and starting with the non-batched operation, an enqueue is linearized in Line 5 in Listing 1, where the next pointer of the last node in the underlying list is modified from `NULL` to point to the new node that the enqueuer created. To show that the abstract state of the queue reflects the change, i.e. the new item is appended to the abstract queue's items, we rely on the following observations:

Observation 8.1. *The next field of a node may change only once in the algorithm, from `NULL` to a pointer to a linked node.*

Proof. During *FutureEnqueue* method (Listing 8), a node intended to be enqueued is thread-local. Its next field's value is initially `NULL`. It might be set to a non-`NULL` value in Line 93, and right after that the local queue's tail is advanced to the new linked node, so the current node will not be locally further modified. The only additional modifications of a node's next field are performed in Lines 5 and 44 (in Listings 1 and 5 respectively), where it might be modified (using a `CAS` operation) from `NULL` to a non-`NULL` value, thus such `CAS` may succeed only once per node. □

Corollary 8.2. *Linking nodes twice to the same node is impossible.*

Observation 8.3. *Consider the shared queue's underlying list of nodes, starting with the initial dummy node, i.e., the node with which the queue is initialized. SQTail always points to a node that is contained in this list. (We ignore the nodes' memory reclamation for simplicity.)*

Proof. The claim holds initially since *SQTail* is initialized to point to the first dummy node, when it is the only node in the list. Based on Observation 8.1, the only changes applied to the queue's list of nodes are additions of nodes to its end. *SQTail* is updated in Lines 7, 13, 50 and 52 inductively from pointing to one node of this list to another, so the claim prevails. □

Consider a successful `CAS` in Line 5 of the next field of the node pointed to by the obtained tail. According to Observation 8.3, this node (which was pointed to by *SQTail*) was part of the list of nodes starting with the initial node of the shared queue, and remains part of this list since, based on Observation 8.1, nodes are not removed from this list. In addition, the previous value of the above mentioned next pointer is `NULL`. Hence, this node, to which we link the new node, is the *last* node in the underlying list of nodes. Specifically, it means that the abstract head points to this node or a prior node. Thus, the node which we link to it becomes one of the nodes in the list that starts with the node succeeding the dummy node (pointed to by the abstract head), which means its item becomes part of the abstract queue in the linearization moment of the enqueue operation. Moreover, the next pointer of the enqueued node is `NULL`, thus only this node's item is appended to the abstract queue's items in this linearization point.

Lastly, similarly to an enqueue operation, all enqueues and dequeues of a batch operation that enqueues at least one item are linearized one after another when the next pointer of the last node in the shared queue's underlying list of nodes is modified from `NULL` to point to the first node of the batch in Line 44 in Listing 5. The abstract state of the queue changes accordingly: The batch's items are appended to the abstract queue's items (the new items become part of the abstract queue due to similar arguments to the ones stated above for a single enqueue). In addition, items that the batch operation dequeues are omitted from the abstract queue in the linearization point, since from this moment the abstract head points to the new dummy node (as defined in Section 8.1.2). This advancing of the head takes into account all enqueues and dequeues of the batch operation, using a calculation described in Section 5.2.

We move to argue about operations that have no effect on the shared queue. Linearization points that do not modify the abstract state of the queue occur during a single dequeue and a dequeues-only batch that are applied to an empty queue. We prove that these linearization points take place when the abstract queue is indeed empty, thus the operations follow the sequential specification of a queue: they appropriately fail and return without affecting the state of the abstract queue. We will focus on a failing dequeue operation, and the same arguments apply for a dequeues-only batch applied to an empty queue.

Let t_{lin} be the linearization moment of an unsuccessful dequeue performed by a thread T, i.e. the moment T executes Line 17, reading the next pointer of its obtained dummy node before revealing in the next line that it is `NULL`. Let t_{read} be the moment in which T executed Line 24 for the last time before t_{lin}. We will prove that the abstract queue is empty at t_{lin}.

The next pointer of the node pointed to by the head obtained at t_{read}, is NULL at t_{lin} (according to t_{lin}'s definition). From Observation 8.1 we deduce that this next pointer was NULL also at any point earlier than t_{lin}, in particular at t_{read}.

At t_{read}, T obtains the queue's head from *SQHead* when no announcement is installed. Thus, the abstract head, according to Section 8.1.2, is either equal to the obtained head, or was equal to it at some previous moment after which a concurrent dequeues-only batch has advanced the abstract head. But the latter is impossible, because for a dequeues-only batch to succeed advancing the head, there must have been at least one node linked after the obtained head, but its next is NULL as mentioned above. Thus, the abstract head equals the obtained head at t_{read}.

The abstract head is not modified between t_{read} and t_{lin}: in order for it to change by a batch operation or a successful dequeue operation, nodes should have been linked – between t_{read} and t_{lin} – after the obtained head, but they have not, based on Observation 8.1. Thus, at t_{lin}, the abstract head still equals the obtained head. Since the next field of the dummy node pointed to by this head is NULL at t_{lin}, the abstract queue is empty at this moment.

8.1.4 No Recurring Linearization Points. We prove why each operation takes effect once, namely, its linearization point occurs one time only. For each linearization point, the thread that performs it does not take any backward branches after that, and the operation completes without repeating the linearization point. We will now establish that other threads do not perform the linearization step again. A successful dequeue operation and a dequeues-only batch that succeeds to dequeue at least one item are achieved using a single operation on the shared queue, thus no help is involved and the initiator thread is the only one to perform the linearization step. An enqueue operation may be assisted by other threads advancing the tail, but the linearization step is taken by the enqueuer only, so no helping thread may perform it and cause the operation to take effect twice. This is not the case for a batch operation that contains enqueues: a helping thread may perform its linearization step. Next we explain how we prevent the linearization step of such operation from occurring twice. The proof is also illustrated in Figure 2.

Let *batchOp* be a batch operation that contains at least one enqueue. The thread that initiates the batch installs an announcement in *SQHead* (in Line 33 in Listing 4), which makes the batch public. Let t_{lin1} be the first time in which *batchOp*'s linearization step is performed (i.e., the CAS in Line 44 in Listing 5 that links the batch's items to the tail is performed successfully for the first time for *batchOp*). Let T_1 be the thread that performed this linking. T_1 could either be the batch's initiator (as depicted in Figure 2) or a helping thread that encountered the announcement. Let t_{read1} be the last time in which T_1 obtained $SQTail$'s value in Line 40 before performing the linearization step,

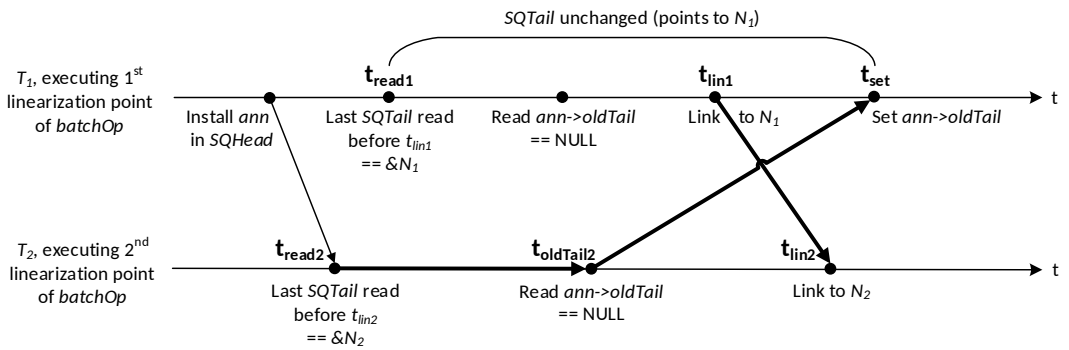

Fig. 2. The impossible scenario of two linearization points of the same batch

and N_1 be the node pointed to by this value (i.e., the node to which *batchOp*'s items are linked at t_{lin1}). Let t_{set} be the first time that Line 47, which sets the *oldTail* field of the announcement, is executed for *batchOp*. It could be executed by any thread; its execution by T_1 in Figure 2 is merely an example. $t_{set} > t_{lin1}$ since Line 47 is executed only after the batch's items are linked.

Assume a second linearization step of *batchOp* is carried out by a thread T_2 at t_{lin2}. Let t_{read2} be the last time in which T_2 obtained *SQTail*'s value in Line 40 before performing the linearization step, and N_2 be the node pointed to by this value (i.e., the node to which T_2 linked *batchOp*'s items at t_{lin2}). After t_{read2} and before t_{lin2}, T_2 obtains the *oldTail* field of the announcement in Line 41 at moment $t_{oldTail2}$. For T_2 to proceed to the linking, this field must be revealed (in Line 42) to be `NULL`.

To complete the proof that a linearization of a batch operation that contains enqueues does not occur twice, we proceed to show how assuming a second linearization point results in a contradiction. This is an overview of the rest of the proof in a nutshell: *SQTail* points to N_1 during t_{read1} through t_{set} (Claim 8.8), thus N_2 (the node pointed to by *SQTail* at t_{read2}, which happens before t_{set}) is either N_1 or a preceding node (Corollary 8.10), hence N_2->*next* is not `NULL` after t_{lin1}, so the `CAS` at t_{lin2} cannot succeed.

To prove that *SQTail* points to N_1 during t_{read1} through t_{set}, we need to establish that *SQTail* does not change in this time frame. To prove that, we will rely on the following lemma:

Lemma 8.4. *For any `CAS` operation of SQTail that occurs between t_{read1} and t_{set}, the previous value passed to the `CAS` is not a pointer to N_1.*

Proof. We list all code lines that modify *SQTail* and explain why the claim holds for each of them.

In Line 7, an enqueue operation advances *SQTail* to point to the node it has just linked. *SQTail* could not have pointed to N_1 prior to this change due to Corollary 8.2, as *batchOp*'s items are linked to N_1. So a `CAS` in Line 7 with a pointer to N_1 passed as the previous value is impossible.

In Line 13, an enqueue operation attempts to assist a conflicting operation and advance *SQTail* using a `CAS`. Suppose that the previous *SQTail*'s value passed to the `CAS` operation is a pointer to N_1 (see Figure 3). We will show that this `CAS` must happen after t_{set}, so it particularly cannot take place between t_{read1} and t_{set}. If the enqueue operation reached Line 13, it means that previously the `CAS` in Line 5 that attempted to link a node to N_1 has failed, then *SQHead* has not consisted of an announcement (when reading *SQHead* in Line 9). For the `CAS` in Line 5 to fail, it must have happened after t_{lin1} (due to Observation 8.1), which in turn happened after the installation of *batchOp*'s announcement in *SQHead*. Thus, *ann* must have been uninstalled from *SQHead* before Line 9's execution. *SQHead* that consists of an announcement is modified only in Method *UpdateHead* (in Listing 5), which is called in Line 53 during the batch's execution. Therefore, Line 53, which

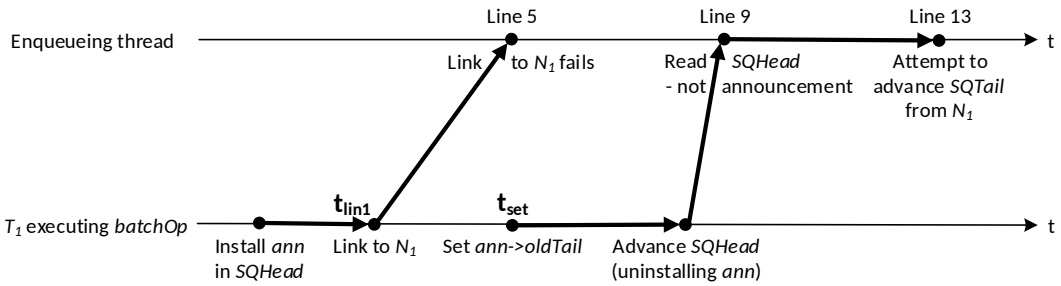

Fig. 3. Enqueuer attempts to advance *SQTail* from N_1

uninstalls the announcement from *SQHead* and completes the *batchOp*'s execution, must have been executed for *batchOp* before Line 9's execution. Prior to the batch completion, as part of *batchOp*'s execution, *ann->oldTail* was set at t_{set}. It follows that the above mentioned CAS in Line 13 happens after t_{set}, which is what we aimed to prove.

Another CAS of *SQTail* in attempt to assist a conflicting operation occurs in Line 50, by a thread, denoted *batchThread*, while it is trying to commit a batch operation. Assume *batchThread* reaches Line 50 with N_1 as the previous value. For this to happen, its attempt to link an item to N_1 in Line 44 must fail, which means N_1's next field is not NULL at that moment. In addition, N_1's next field does not point to the first node enqueued by the batch that *batchThread* executes, according to the check in Line 45. Consequently, N_1's next field must point to another node, linked by an operation denoted $conflictingOp$, which is conflicting with the batch operation that *batchThread* executes. We will show that $conflictingOp$ is not *batchOp*, and thus this scenario is impossible – due to N_1's definition as the node to which *batchOp*'s items were linked, and based on Corollary 8.2. So it remains to show that $conflictingOp$ is not *batchOp*: *batchThread* tries to commit either *batchOp*, or another batch operation, which we will denote by *batchOp*2. We will cover both cases. If *batchThread* is trying to carry out *batchOp*, then failing the check in Line 45 clearly indicates, as mentioned above, that an operation which is not *batchOp* has linked a node to N_1. If *batchThread* is trying to carry out *batchOp*2, then according to Claim 8.5 (which is brought after this proof), $conflictingOp$ is a single enqueue, and in particular not *batchOp*.

An additional modification of *SQTail* happens during a batch execution in Line 52. Suppose some thread advances *SQTail* in Line 52, and suppose that the previous value passed to the CAS operation is a pointer to N_1. If the thread tries to carry out *batchOp*, it does not advance the tail between t_{read1} and t_{set}: to reach Line 52 it has to break from the while loop, which could happen only after t_{set} (the first time *oldTail* field of *batchOp*'s announcement was set). Otherwise, the thread tries to carry out another batch operation. To reach Line 52 it has to break from the while loop. This happens only after Line 47 is carried out for this other batch and sets the *oldTail* field of the batch's announcement to point to N_1. This, in turn, happens only after the first node that the other batch wishes to enqueue has been linked to N_1 (according to Line 45). But this is impossible, due to N_1's definition as the node to which *batchOp*'s items were linked, and based on Corollary 8.2. □

We shall prove the following claim and lemmas to complete the last proof:

Claim 8.5. *When attempting to advance the tail in Line 50 in ExecuteAnn after an attempt to link a batch's items to a node N has failed, the conflicting operation that linked to N is necessarily a single enqueue and not a batch operation.*

Proof. See Figure 4 for an illustration of the proof. Let *batchOp*1 be a batch operation containing at least one enqueue. Let T_1 be a thread that attempts at t_{fail} to link *batchOp*1's items to N, the node it obtained from *SQTail*, and fails. T_1 then checks in Line 45 whether another thread has linked *batchOp*1's items to N and caused T_1 to fail performing the CAS. Assume T_1 finds out that this is not the case, i.e., an item of another operation has been linked to N. Hence, T_1 needs to take a backward branch. Assume the conflicting operation whose root step caused T_1's CAS to fail is another batch operation that contains at least one enqueue, denoted *batchOp*2. We will show that this assumption leads to a contradiction, hence the conflicting operation must be a single enqueue.

Let T_2 be the thread that executed the above mentioned root step, linking *batchOp*2's items to N at t_{lin2} prior to t_{fail}. Let H be the history described in the proof (and in Figure 4). Let t_{read} be the moment in which T_1 obtained the *oldTail* field of *ann*1, *batchOp*1's announcement, in Line 41. The obtained value must be NULL since T_1 proceeded to a linking attempt. We will establish in Lemma

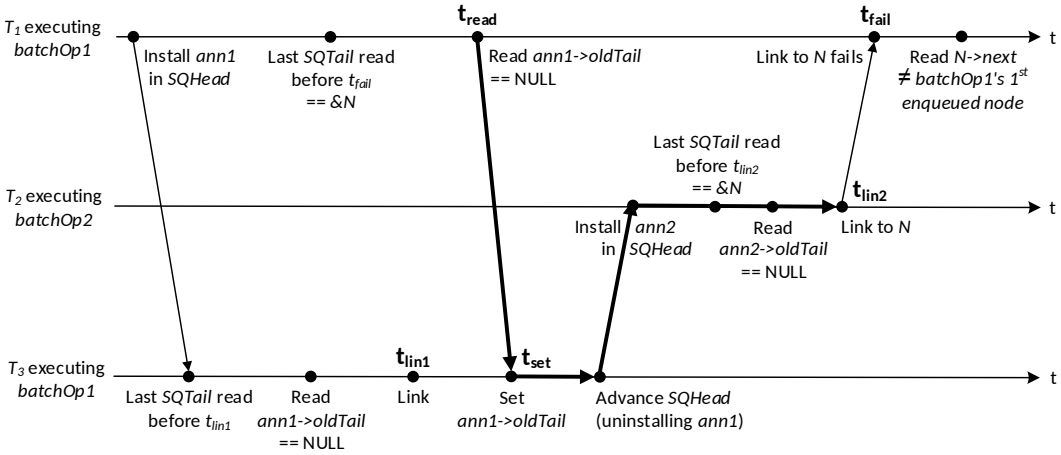

Fig. 4. The impossible scenario of one batch failing another

8.6 that $t_{read} < t_{lin2}$, so at $t_{lin2} - \epsilon$, T_1 has passed Line 41. It hasn't passed Line 44, since in H it executes this line at t_{fail}, which happens after t_{lin2}. So at $t_{lin2} - \epsilon$, $T1$'s next step is either Line 42 or 44.

Now, consider an alternative history H', which starts with the same prefix as H until $t_{lin2} - \epsilon$, but then T_1 is scheduled to run rather than $T2$. The result of T_1 executing Line 42 is predetermined: the value obtained from *ann*1->*oldTail* is `NULL`, as previously mentioned. So whether T_1 executes this line in H' at $t_{lin2} - \epsilon$ or earlier, it will proceed to attempt a `CAS` in Line 44, trying to link *batchOp*1's items to N. This `CAS` would succeed, since in H, T_2 succeeded performing a `CAS` of the tail at the same moment, which means that the current value of the tail's next pointer must be `NULL`.

To reach a contradiction, we will prove in Lemma 8.7 that a linearization step has been carried out for *batchOp*1 in H at t_{lin1}, prior to t_{lin2}. Therefore, it was carried out in H' as well. This implies that two linearization steps have been performed for *batchOp*1 in H': both at t_{lin1} and in the new suffix of H'. This contradicts what we proved in Section 8.1.4 – that no operation has two linearization points – and concludes our proof. □

Lemma 8.6. *t_{read} happens before t_{lin2}.*

Proof. For T_2 to link *batchOp*2's items at t_{lin2}, an announcement *ann*2 for this batch must first be installed in *SQHead* (either by another thread, or by T_2 as illustrated in Figure 4). This must happen after the installation of *ann*1, because if *batchOp*2 happened before *batchOp*1, then before *ann*2 was uninstalled, the tail had been advanced from pointing to N, so T_1 could not obtain N from the tail while executing *batchOp*1. Clearly, prior to *ann*2's installation, *ann*1 must be uninstalled from *SQHead* (by a thread executing *batchOp*1, which could be another thread T_3 as depicted in Figure 4, and could also be T_2 itself, helping completing *batchOp*1's execution). Before *ann*1 is uninstalled as the last step of *batchOp*1's execution, *ann*1->*oldTail* is set as part of the batch's execution at t_{set} (by T_3 or by another thread executing *batchOp*1). So far we showed that $t_{set} < t_{lin2}$. In addition, t_{set} must happen after t_{read}, when the value of *ann*1->*oldTail* is still `NULL`. □

Lemma 8.7. *batchOp*1*'s items were linked at t_{lin1} earlier than t_{lin2}.*

Proof. Before *ann*1->*oldTail* is set at t_{set}, *batchOp*1's items must have been linked as part of the batch's execution (by T_3 or by another thread executing *batchOp*1), at moment $t_{lin1} < t_{set}$. In addition, in the proof of Lemma 8.6 we argued why $t_{set} < t_{lin2}$. Consequently, $t_{lin1} < t_{lin2}$. □

We continue with the proof that assuming a second linearization point results in a contradiction.

Claim 8.8. *$SQTail$ is not modified between t_{read1} and t_{set}.*

Proof. At t_{read1}, $SQTail$ points to N_1 (by t_{read1}'s definition). According to Lemma 8.4, no `CAS` of $SQTail$ may be the first to modify it from N_1 between t_{read1} and t_{set}. Thus, no successful `CAS` of $SQTail$ occurs during this time frame. □

Lemma 8.9. *Up to t_{set}, $SQTail$ points to either N_1 or a preceding node.*

When mentioning a *preceding* or *subsequent* node, we refer to the nodes' order in the queue's underlying list of nodes. We view this list as starting with the initial dummy node, so it contains all nodes that were ever enqueued. (In the proof we ignore the nodes' memory reclamation for simplicity, but anyhow in practice the threads do not hold pointers to reclaimed nodes.)

Proof. According to Observation 8.3, $SQTail$ always points to a node in the queue's underlying list of nodes. Up to t_{lin1}, this list does not consist of any nodes subsequent to N_1, so $SQTail$ must point to N_1 or a preceding node. Namely, up to t_{lin1} the claim holds. In view of Claim 8.8, $SQTail$ remains the same since t_{read1}, and in particular since t_{lin1}, until t_{set}. Hence, the claim prevails. □

Corollary 8.10. *N_2, the node pointed to by the tail obtained by T_2 at t_{read2}, is either N_1 or a preceding node.*

Proof. T_2's reading of $SQTail$ at t_{read2} happens before t_{set}, because after t_{read2}, the announcement's *oldTail* field is still `NULL` at $t_{oldTail2}$. The claim immediately follows from Lemma 8.9. □

Claim 8.11. *At t_{lin2}, N_2->$next \neq$ `NULL`.*

Proof. $t_{lin2} > t_{lin1}$ based on t_{lin1}'s definition as *batchOp*'s first linearization point. Hence, by the time of t_{lin2}, T_1 has linked a node to N_1. Nodes had been clearly previously linked to all preceding nodes as well. According to Corollary 8.10, N_2 is either N_1 or a preceding node, so a node has been linked to it before t_{lin2}. This implies that the *next* field's value of N_2 is not `NULL` at t_{lin2}. □

Claim 8.11 yields a contradiction to the assumption of a second linearization point, as the `CAS` at t_{lin2} is destined to fail.

8.2 Lock-Freedom Proof

In this subsection we show that our algorithm ensures system-wide progress. In the algorithm of BQ, announcements are used to assist in constituting lock-freedom: a thread that wishes to perform a batch operation installs an announcement describing the batch in the shared queue's head. The purpose of the installation is to enable other threads to complete this batch operation so that they can thereafter proceed to perform their own operations, even if the thread that installed the announcement is delayed.

To prove that our algorithm is lock-free, we break each of the shared queue's operations down to a sequence of intermediate progress steps.

Definition 8.12. The completion of an interface method of the queue (one of *Enqueue*, *Dequeue*, *FutureEnqueue*, *FutureDequeue* and *Evaluate*) is labeled a *full progress step.*

The following operations may be applied to the shared queue: enqueue, dequeue, batch with at least one enqueue and dequeues-only batch. We will refer to them as the *shared queue's operations.* The methods *EnqueueToShared*, *DequeueFromShared*, *ExecuteBatch* and *ExecuteDeqsBatch* apply these operations respectively.

Definition 8.13. An *intermediate progress step* is a `CAS` operation that achieves progress toward achieving a full progress step. It might be executed either by the thread that initiated the operation or by a helping thread. It is a point of no return in the context of the current shared queue's operation: once a thread (either the initiator or a helping thread) that executes a shared queue's operation detects that an intermediate progress step has been completed for this operation, it may not branch back to a step in that shared queue's operation that is earlier than the completed intermediate progress step.

Definition 8.14. A *backward branch* refers to branching back to an earlier point in the execution of the same shared queue's operation, due to a step, carried out by an obstructing operation, that prevents the current operation from achieving its pursued intermediate progress step.

Definition 8.15. An intermediate progress step s is the *root step* of a backward branch b, if s prevents the thread that executes b from achieving an intermediate progress step, which causes this thread to take the backward branch b after revealing the obstruction.

Note that all backward branches are caused by conflicting intermediate progress steps, so a root step is necessarily an intermediate progress step.

In practice, after detecting an obstructing step and before the jump backwards, an attempt to assist an obstructing operation to complete might be made. Afterwards, there is a branch back to the beginning of the current loop - a loop that appears right after the last intermediate step, or at the beginning of the operation if no intermediate progress step has been accomplished yet. The thread then starts another iteration in pursue of accomplishing the same intermediate progress step.

Observation 8.16. *These are the intermediate progress steps of the shared queue's operations:*

(1) In EnqueueToShared method:
- *A `CAS` of the next pointer of the node pointed to by the shared queue's tail from `NULL` to the enqueued node.*
- *A `CAS` of the shared queue's tail from the current tail to a pointer to the enqueued node.*

(2) In DequeueFromShared method: In case the next field of the node pointed to by the obtained queue's head is `NULL`, DequeueFromShared finishes and returns `NULL` without performing any intermediate progress steps. Otherwise:
- *A `CAS` of the shared queue's head from the current head to a pointer to the next node.*

(3) In ExecuteDeqsBatch method: In case the next field of the node pointed to by the obtained queue's head is `NULL` – ExecuteDeqsBatch finishes without performing any intermediate progress steps. Otherwise:
- *A `CAS` of the shared queue's head from the current head to a pointer to the last node dequeued by the batch operation.*

(4) For a batch with at least one enqueue (The first intermediate progress step is performed in ExecuteBatch method and the rest are performed in ExecuteAnn auxiliary method):
- *A `CAS` of the shared queue's head from the current head to the batch's announcement.*
- *A `CAS` of the next field of the node pointed to by the shared queue's tail from `NULL` to a pointer to the first node enqueued by the batch operation.*
- *A `CAS` of the shared queue's tail from the current tail to a pointer to the last node enqueued by the batch operation.*
- *A `CAS` of the shared queue's head from the installed announcement to a pointer to the last node dequeued by the batch operation.*

This is the outline of the lock-freedom proof: We examine the execution from a given moment t. We need to show that a full progress step is achieved in a finite number of steps. Each intermediate

progress step could cause a bounded number of backward branches (Lemma 8.19), and each thread performs a finite number of steps in any execution segment that contains no backward branches (Observations 8.20 and 8.22). Based on this, we prove that every finite number of steps, a progress step – intermediate or full – is achieved (Lemma 8.23). If a full progress step is achieved, we are done. As long as this does not happen, intermediate progress steps keep being achieved, and after a bounded number of them – a full progress step is eventually achieved (Lemma 8.24).

Claim 8.17. *Let T be a thread executing a shared queue's operation. Suppose T takes a backward branch, after it detects a root step performed by another thread. Then the same root step can cause only one additional backward branch in the same code line in T's run.*

Proof. We list all backward branches and show that, as claimed, each of them is caused by a simultaneous conflicting operation that would cause at most one additional backward branch when the same thread executes the same line later.

Let T be a thread executing an operation on the queue. We review the backward branches according to the shared queue's operations. Note that in addition to the backward branches T might take while trying to commit intermediate progress steps of its current operation, T might encounter conflicting operations and take backward branches also while assisting them and trying to commit their intermediate progress steps.

1. Enqueue (performed by *EnqueueToShared*, Listing 1): While executing an enqueue operation, T obtains the value of the tail in Line 4. Let N be the node pointed to by this value. If T then fails to `CAS` the next pointer of N (in Line 5), it is due to a conflicting operation that has linked a node to N in a step denoted s. As a result of the `CAS` failure, T attempts to assure the tail is advanced, and then takes a backward branch due to s. The conflicting operation could be either an enqueue or a batch.

1.1. First, we analyze the case of a single enqueue operation $enq1$ executing the linking step s. We will show that after at most two backward branches of T due to linking failures, the tail no longer points to N (as either T or another thread has advanced the tail). Thus, afterwards, when T attempts to link to the tail, it shall not fail again due to $enq1$.

1.1.1. If when T obtains the head (in Line 9) it does not consist of an announcement, then T tries to advance the tail to point to the node linked by s. If this attempt fails, it implies that the tail has been already advanced by another thread. So in this case, the tail no longer points to N after a single backward branch.

1.1.2. On the other hand, the head value obtained by T might consist of an announcement. Let t_{fail1} be the moment T fails to `CAS` the next pointer of N when trying to perform an enqueue operation denoted $enq2$, and t_{lin} be s's execution moment, namely, the moment in which a thread denoted T_2 links a new item to N while executing a conflicting enqueue operation $enq1$ (different from $enq2$). We assumed that when T reads the head after the `CAS` fails, it obtains a value pointing to an announcement. Let *ann* be this announcement, installed for a batch operation $batchOp$. Right after reading the head, T calls *ExecuteAnn* to assure the completion of $batchOp$'s execution.
Let t_{read} be the moment in which T_2 obtains the tail value for the last time before t_{lin}. $batchOp$'s step of advancing the tail could either happen before or after t_{read}, with the implication of $batchOp$ happening before $enq1$ or vice versa.

1.1.2.1. The following scenario is illustrated in Figure 5. Suppose $batchOp$'s step of advancing the tail has been accomplished (by the batch initiator as illustrated in Figure 5 or by a helping thread) before t_{read}. I.e., $batchOp$ has linked its items before $enq1$ has linked its item. In this scenario, a root step may cause *two* backward branches of the same thread in the same code line.

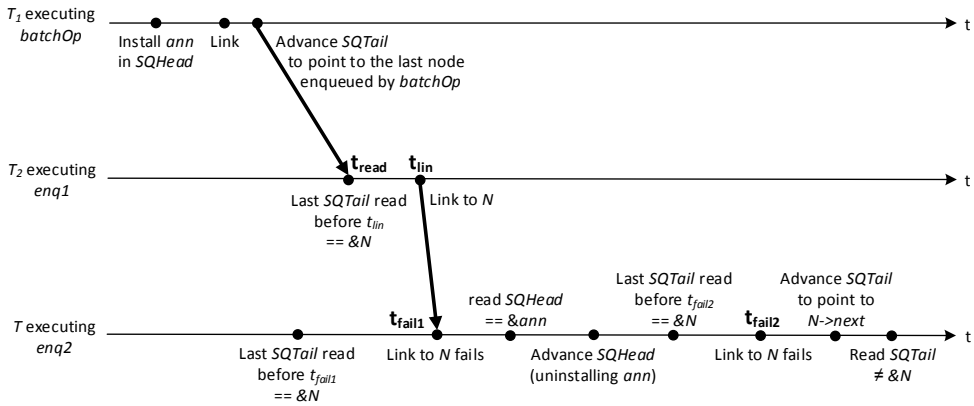

Fig. 5. *enq*1 causes *enq*2 to fail while *ann* is installed in the head, after *batchOp* has advanced the tail

The tail has already been advanced for *batchOp* when T executes *ExecuteAnn* method (since this happens after t_{fail1}, the tail is advanced before t_{read}, and $t_{read} < t_{lin} < t_{fail1}$). Therefore, the only remaining step *ExecuteAnn* applies to the shared queue is uninstalling *ann* from the head if it has not been uninstalled yet. All previous steps of the batch execution are already done. Then, T branches backwards to start a second linking attempt. It is possible that no thread has advanced the tail yet from pointing to N. In such a case, T would obtain a pointer to N again when reading $SQTail$, and would again fail to link to N. The difference from the previous loop iteration is that now *ann* is no longer installed in the head. When T reads the head in Line 9, it either points to a node or to an announcement. In the first case (no announcement installed in the head), like in case 1.1.1., T makes sure in Line 13 that the tail is advanced. In the second case, a new announcement is installed, and like in case 1.1.2.2., its batch happens after *enq*1. T makes sure this batch is completed by calling *ExecuteAnn*. During this call - prior to linking the batch's items - the tail is advanced in Line 50 from pointing to N if it has not been advanced earlier. The reason thaÂŋt this time, unlike when executing *ExecuteAnn* for *ann*, the tail must be advanced, is that the new announcement happens after *enq*1, so it must assist *enq*1 to advance the tail, to subsequently link its own items. Consequently, in any case, when T reads the tail (in Line 4) again after the second linking failure, the tail no longer points to N.

1.1.2.2. On the other hand, *batchOp*'s step of advancing the tail might happen after t_{read}. This implies that *batchOp*'s items are linked to a node down the list, subsequent to N, after t_{lin} (because if they were linked before t_{lin}, the tail must have been advanced before t_{lin}, for T_2 to be able to link after the tail). Before linking *batchOp*'s items, the tail must be advanced from pointing to N to point to the next node. Thus, when T's call to *ExecuteAnn* returns after the batch completion, the tail has already been advanced from pointing to N.

1.2. Second, we analyze the case in which a batch operation *batchOp* that contains at least one enqueue is the one to execute the linking step s. We will show that by the time T branches backwards, the tail will have already been advanced from pointing to N to point to the last node enqueued by *batchOp*. Therefore, if T fails to link to the tail again, it would be to a

new tail to which another conflicting operation has linked an item, so s would not cause T to take another backward branch.
If when T obtains the head it does not consist of *batchOp*'s announcement, then *batchOp*'s announcement must have been uninstalled, so *batchOp*'s execution has been completed, including advancing the tail (and if the obtained head is not an announcement, then T would perform a `CAS` of the tail in Line 13 but it would fail). Otherwise, T calls *ExecuteAnn* to assure that *batchOp*'s execution is completed, including advancing the tail.
It remains to explain why any advancing of the tail from N is necessarily to the last node enqueued by *batchOp*, and not to the first one. This is required to guarantee that s is not going to be the root step of additional backward branches of T, which could be the case if the tail were advanced node by node through all *batchOp*'s enqueued nodes. We will prove that the tail cannot be advanced from pointing to N in the two occasions (in Lines 13 and 50) of attempting to advance the tail by one node to assist a conflicting operation, thus it could be only advanced from pointing to N to point to the last enqueued node in Line 52. Claim 8.5 proves that the assisted operation in Line 50 must be a single enqueue. But *batchOp* – and not a single enqueue operation – linked to N, thus $SQTail$ cannot be advanced from pointing to N in Line 50. For the other occasion, of Line 13 (see also Figure 6): T reaches Line 13 after failing to link a node to N in Line 5, and later reading the head when it does not consist of an announcement. For the `CAS` in Line 5 to fail, it must have happened after either *batchOp*'s initiator or an assisting thread has linked *batchOp*'s items to N (due to Observation 8.1), which in turn happened after the installation of *batchOp*'s announcement, denoted *ann*, in the head. Since *ann* has been installed in the head before T executes Line 9, it must have also been uninstalled before Line 9's execution. $SQHead$ that consists of an announcement is modified only in Method *UpdateHead* (in Listing 5), which is called in Line 53 during the batch's execution. Therefore, Line 53, which uninstalls the announcement from $SQHead$ and completes the *batchOp*'s execution, must have been executed for *batchOp* before T executes Line 9. Prior to the batch completion, as part of *batchOp*'s execution, $SQTail$ is advanced. Namely, *batchOp*'s execution has been completed, including advancing the tail, before T read the head and thus before T performed a `CAS` of the tail in Line 13. Hence, this `CAS`, with N as the previous value, would fail.

2. Dequeue (performed by *DequeueFromShared*, Listing 2) has two possible backward branches:
   - *DequeueFromShared* first calls *HelpAnnAndGetHead* auxiliary method. In case *HelpAnnAndGetHead* encounters an announcement installed in the shared queue's head by a conflicting batch operation, it assists the related batch by calling *ExecuteAnn* (in Line 27). Then,

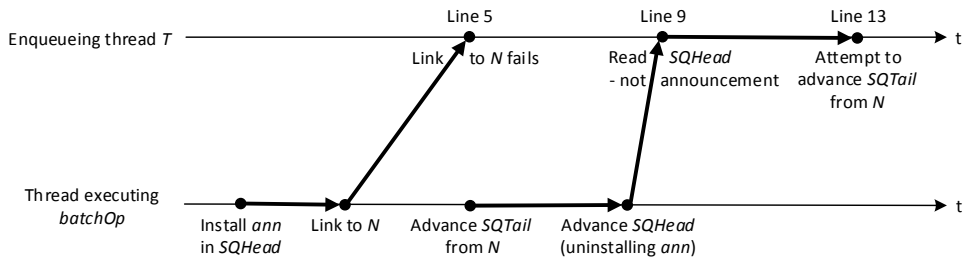

Fig. 6. Enqueuer fails to advance $SQTail$ from N

after the batch's execution is assured to be over, including uninstalling its announcement, the dequeuer branches backwards and will certainly not encounter the same announcement installed in the head again.

- If the `CAS` of the shared queue's head (in Line 20 in *DequeueFromShared*) fails, the dequeuer thread restarts the operation. When the dequeuer attempted to perform the `CAS` from its obtained value N, the head pointed to either a node down the list, or an announcement *ann*. In the first case, a pointer to a node different from N would be obtained the next time the dequeuer obtains the head, thus the dequeuer may not encounter again the same conflicting operation that advanced the head from pointing to N. In the second case, after the dequeuer restarts the operation, it calls *HelpAnnAndGetHead*. If the head that this method obtains does not point to *ann*, then *ann* has been uninstalled and would not cause another backward branch. However, if the obtained head points to *ann*, then the dequeuer assists it to complete (in Line 27) and then branches backwards - for the second and last time due to this batch operation.

3. Dequeues-only batch (performed by *ExecuteDeqsBatch*, Listing 12): Its backward branches are similar to those of *DequeueFromShared*.
4. A batch containing at least one enqueue (performed by *ExecuteBatch*, Listing 4): It may take the same backward branches as *DequeueFromShared*, and one more backward branch - detailed next - during its call to the auxiliary method *ExecuteAnn* (Listing 5) for the current batch (in Line 35).
   ExecuteAnn branches to its beginning in case it fails to `CAS` the next pointer of the node pointed to by the tail, to point to the first enqueued node of the batch (in Line 44), and no other thread has accomplished this modification. Before restarting, *ExecuteAnn* helps the conflicting operation and advances the queue's tail by one node, in case it has not yet been advanced by another thread (in Line 50). If the conflicting operation were a batch, its linearization step could be the root step of several backward branches: *ExecuteAnn* might advance the tail node by node, one node down the list before each backward branch, until the tail would point to the conflicting batch's last enqueued node. But this is impossible, since the conflicting operation must be a single enqueue and not a batch operation, according to Claim 8.5. Thus, advancing the tail by one node completes the conflicting operation. After *ExecuteAnn*'s restart, a new tail would be obtained, and the same root step would not cause a second `CAS` failure in Line 44.

□

Assumption 8.18. *There is a bounded number of threads operating simultaneously on the shared queue, denoted n.*

We denote the number of code lines in which backward branches may occur by B.

Lemma 8.19. *Each intermediate progress step may be the root step of up to* $2B(n-1)$ *backward branches.*

Proof. From Claim 8.17, it follows that a root step may cause at most 2 backward branches per backward branch code line per thread. Moreover, a root step may not cause a backward branch in the thread that carried it out, since this thread is aware of the step and will not attempt to apply conflicting operations. □

Observation 8.20. *Each queue's interface method, denoted IM, wraps zero or one internal method that applies a shared queue's operation. Other than the call to this internal operation method, IM*

executes $O(pendingOps)$ computational steps, where pendingOps is the number of pending operations in the thread executing the interface method.

Proof. *Enqueue* calls either *EnqueueToShared*, or *Evaluate* that in turn calls an internal method as detailed next. *Dequeue* calls either *DequeueFromShared* or *Evaluate*. *Evaluate* calls either *ExecuteBatch* or *ExecuteDeqsBatch*. *FutureEnqueue* and *FutureDequeue* do not call any of the queue's internal methods.

Other than these calls to internal methods, all interface methods execute $O(1)$ computational steps (with no backward branches), with the exception of the *Evaluate* method. This method, called either by the user or by *Enqueue* or *Dequeue*, calls *PairFuturesWithResults* or *PairDeqFuturesWithResults*. These result-pairing methods make $O(pendingOps)$ computational steps. □

Definition 8.21. A *forward segment* is a maximal part of a method's execution that contains no backward branches. Namely, if the method did not take any backward branches, then its whole execution is a single forward segment, otherwise its execution is composed of a first forward segment – from its invocation until the first backward branch, possible forward segments between each two consecutive backward branches performed throughout this method's execution, and a last forward segment – from after the last backward branch until the method's response.

Observation 8.22. *Each shared queue's operation method makes $O(pendingDeqs)$ computational steps in each of its forward segments, where pendingDeqs is the maximum number of pending dequeue operations in any of the threads during the method's execution.*

Proof. Each of *EnqueueToShared*, *DequeueFromShared* and *ExecuteBatch* performs $O(1)$ computational steps in each of its forward segments, except for when calling *ExecuteAnn* (directly or through a call to *HelpAnnAndGetHead*). *ExecuteAnn*, called to commit the batch operation of the caller thread or an assisted thread, performs $O(1)$ computational steps in every forward segment, except for its last forward segment, in which it carries out $O(pendingDeqs)$ computational steps during the call to *GetNthNode* auxiliary method (in Line 64 or 66) that calculates the new head. Regarding *ExecuteDeqsBatch*, every time it calls *ExecuteAnn* during a call to *HelpAnnAndGetHead*, it performs forward segments of $O(pendingDeqs)$ computational steps as detailed above. The rest of *ExecuteDeqsBatch*'s execution is also made of forward segments of $O(pendingDeqs)$ computational steps, each due to a traversal of the dequeued nodes when calculating the new head (in Lines 152-157). Thus, in any case, $O(pendingDeqs)$ computational steps are carried out in every forward segment of a shared queue's operation method. □

Lemma 8.23. *From each moment t, some intermediate or full progress step is accomplished within a finite number of system-wide computational steps (of methods operating on the queue).*

Proof. Let *IPS* be the total number of intermediate progress steps achieved in the execution until t, and *pendingDeqs* and *pendingOps* be the maximum number of pending dequeue operations and pending operations of any kind (enqueue or dequeue) respectively in any of the threads at t.

Since moment t, as long as no progress step – intermediate or full – is achieved by any thread: There are $O(IPS \cdot B \cdot n)$ backward branches across all threads, based on Lemma 8.19. From Observation 8.22, each thread performs $O(pendingDeqs)$ computational steps in each of its forward segments (i.e., between each two consecutive backward branches it takes, as well as before the first and after the last). (Note that as long as no progress steps are achieved since t, the number of pending operations in each thread remains as it was at t, hence *pendingDeqs* remains a bound on the number of pending dequeue operations in any of the threads.) It follows that after $O(IPS \cdot B \cdot n \cdot pendingDeqs)$ computational steps, there could be no more backward branches, so each thread that runs an internal method must return to the calling interface method. According to Observation 8.20, each thread

performs $O(pendingOps)$ additional steps before returning from the executed interface method, i.e., performing a full progress step. (Note that a new pending operation could not be formed in a thread after t before it performs a progress step, hence *pendingOps* remains a bound on the number of pending operations in any of the threads.) Thus, overall, there are additional $O(n \cdot pendingOps)$ steps before a full progress step is achieved. Such full progress step is guaranteed to happen, due to the arguments we laid out, as long as no progress step is achieved beforehand. But if a thread performs a progress step earlier, we are anyhow done. Thus, in any case, an intermediate or full progress step is achieved within $O(IPS \cdot B \cdot n \cdot pendingDeqs + n \cdot pendingOps)$ steps since t. This is sufficient – there is no need for the tightest bound, as all we need to show is that progress is achieved within a *finite* number of steps.

□

Lemma 8.24. *Every up to $4n$ system-wide intermediate progress steps, a new full progress step is accomplished.*

Proof. Each queue's interface method executes intermediate progress steps only during calls to shared queue's operations. According to Observation 8.20, every interface method calls at most one shared queue's operation. Each shared queue's operation is composed of up to 4 intermediate progress steps (see Observation 8.16). Therefore, while the threads execute methods operating on the queue, after at most $4n$ intermediate progress steps, all threads will return from their executed interface methods before another intermediate progress step is accomplished. □

Corollary 8.25. *BQ is lock-free.*

Proof. We need to prove that from any moment, a queue's interface method is completed within a finite number of steps. Lemma 8.23 states that within a finite number of steps, an intermediate or full progress step is accomplished. If it is a full progress step, we are done. Otherwise, we apply Lemma 8.23 repeatedly at most $4n$ times, and get from Lemma 8.24 that a new full progress step must be eventually accomplished. □

9 PERFORMANCE

We compared the proposed BQ to two queue algorithms: the original MSQ that executes one operation at a time, and the queue by Kogan and Herlihy [17] that satisfies MF-linearizability, hence denoted KHQ. KHQ executes pending operations in batches of homogeneous operations: it executes each subsequence of enqueues-only together by linking nodes to the end of the queue, and each subsequence of dequeues-only by unlinking nodes from the head of the queue.

We implemented the shared parts of the different queue versions identically to filter any unrelated performance difference. All queues use the optimistic access scheme for memory management. The implementations were coded in C++ and compiled using the GCC compiler version 6.3.0 with a $-O3$ optimization level.

We conducted our experiments on a machine running Linux (Ubuntu 16.04) equipped with 4 AMD Opteron(TM) 6376 2.3GHz processors. Each processor has 16 cores, resulting in 64 threads overall. The number of threads in each experiment varied from 1 to 128. Each thread was attached to a different core, except for the experiment that ran 128 threads, in which two threads were attached to each core. The machine used 64GB RAM, an L1 data cache of 16KB per core, an L2 cache of 2MB for every two cores, and an L3 cache of 6MB for every 8 cores.

In each experiment testing BQ or KHQ, our workload performed batch operations with a fixed number of future operations for that experiment. Our workload for MSQ performed standard operations only. In the workloads of all queues, we randomly determined whether each operation (standard or future) would be an enqueue or a dequeue.

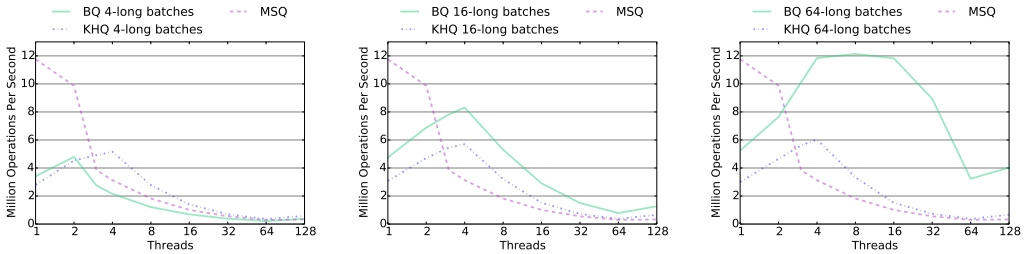

Fig. 7. Throughput for 4, 16 and 64 long batches respectively

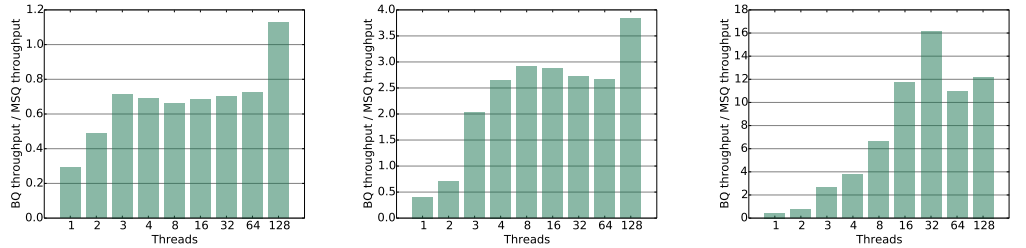

Fig. 8. Throughput ratio of BQ compared to MSQ for 4, 16 and 64 long batches

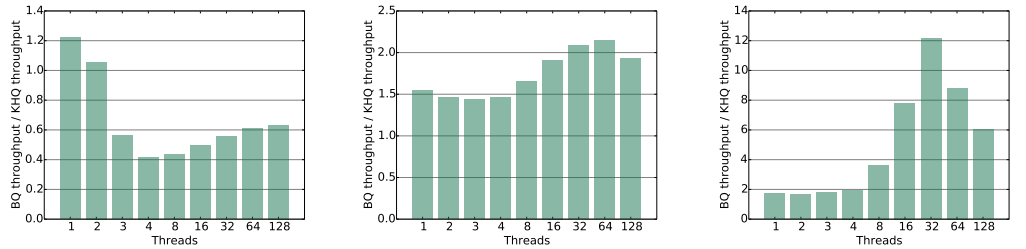

Fig. 9. Throughput ratio of BQ compared to KHQ for 4, 16 and 64 long batches

Each data point $[x, y]$ in the graphs in Figure 7 represents the average result of 10 experiments. In each experiment, x threads performed operations concurrently for two seconds. The graphs depict the throughput in each case, i.e., the number of operations (*Enqueue* / *Dequeue* / *FutureEnqueue* / *FutureDequeue*) applied to the shared queue per second by the threads altogether, measured in million operations per second. The BQ curve appears along with the MSQ and KHQ curves for different batch sizes.

In addition to the throughput graphs of BQ compared to MSQ and KHQ, we display the corresponding throughput ratio graphs. Each chart in Figure 8 shows the ratio between the throughput of BQ for a certain size of batches and the throughput of MSQ. Similarly, each chart in Figure 9 shows the ratio between the throughputs of BQ and KHQ for a certain size of batches. A ratio bigger than 1 means that batching all operations to a single operation was beneficial, and a fraction means that it was detrimental.

BQ demonstrates a significant performance improvement over both competitors for batches of more than 10 operations. Indeed, for batches containing 4 operations, MSQ and KHQ are preferable. The overhead of executing a batch operation makes small batches less worthwhile. However, for longer batches, and when at least 3 threads operate on the queue, BQ performs better.

BQ exploits parallelism better as execution of operations in batches reduces contention substantially: instead of accessing the shared queue for every operation, each thread interacts with the

shared queue throughout the execution of a single batch operation. Later, it performs local work to pair futures applied by the batch operation with results. BQ performs better than KHQ as well, since it applies each batch at once to the shared queue, while KHQ applies each batch operation using several homogeneous batch executions. Therefore, BQ is an excellent choice for a lock-free queue when future operations can be employed.

The throughput of MSQ decreases as the number of threads increases, since the contention makes it impossible to exploit parallelism. On the other hand, when using batches of size 16 or more, 3 threads achieve better throughput in BQ than 2, and 4 perform better than 3, demonstrating improved scalability.

The more operations a batch contains, the greater the performance gap between BQ and the other queues becomes. BQ performs better as batch size increases since the reduction in contention more than compensates for the greater overhead. The performance improvement from batches containing 4 operations to ones containing 16, and from these to 64-long batches, is shown in Figure 7. Additional measurements we conducted demonstrate further improvement for 256-long batches in comparison to 64-long batches, especially for executions with many threads.

10 AVOID USING DOUBLE-WIDTH CAS

To make the algorithm portable to platforms that do not implement double-width CAS, the algorithm may be modified to use a single-word CAS only. Currently, *SQHead* is a *PtrCntOrAnn* object and *SQTail* is a *PtrCnt* object. Both of them are double-word wide, so an atomic update of them requires a double-width CAS. To avoid it, *SQHead* and *SQTail* may become pointers only.

The dequeue and enqueue counters associated with the head and tail respectively are still required, because we use them to calculate the queue's size when a batch is operated in order to figure out the new head. We will place a counter in the *Node* object, to indicate the node's index in the shared queue (i.e., its index in the underlying list of nodes starting with the node following the initial dummy node). Right before the queue's head or tail is updated to point to a certain node, we will set the node's counter. This occurs in the following cases:

(1) A single dequeue operation will update *head->next.count* before performing a CAS of the head. Similarly, a dequeues-only batch operation will update the counter of the node pointed to by the new head before performing a CAS of the head to point to this node. The new counter value in this case is the amount of dequeues in the batch accumulated to the counter of the node currently pointed to by the head.
(2) A single enqueue operation will update the counter of the new node before linking it to the tail.
(3) When a thread carries out a batch operation that contains at least one enqueue operation, it will update the counter of its last enqueued node, which is about to be pointed to by the new tail. The new counter value equals the amount of enqueues in the batch summed to the counter of the node pointed to by the current tail. This update shall be performed right before the CAS of the tail to point to the last enqueued node. If other threads try to execute this batch operation simultaneously, they may also perform this update, as the amount of enqueues in the batch is detailed in the announcement.
If the batch operation contains at least one successful dequeue, it will also update the counter of the node that is about to be pointed to by the head, right before performing the head's CAS that completes the batch execution.

Note that writing the counter does not require a CAS: The written value is the node's index in the queue's sequence of items, which is unambiguous. Therefore, under no circumstances may two threads try to write different values.

An additional adaptation is required to distinguish whether *SQHead* points to a node or an announcement. Currently, the least significant bit of $SQHead.tag$, which overlaps $SQHead.ptrCnt.node$, is set to indicate that *SQHead* contains an announcement. In the new suggested design, we would take a similar approach and use the least significant bit of the *SQHead* pointer as a mark.

11 CONCLUSION

We presented BQ, a novel lock-free extension to MSQ that supports future operations. Unlike KHQ, BQ supports single operations as well, according to EMF-linearizability. BQ exploits batching to reduce contention and improve scalability. It enables a fast application of a mixed sequence of enqueue and dequeue operations all at once to the shared queue. Thus, it significantly reduces accesses to the shared queue and overall processing in it, in comparison to both MSQ and KHQ. BQ demonstrates a substantial performance improvement of up to 16x compared to MSQ and up to 12x compared to KHQ.

REFERENCES

[1] Dmitry Basin, Rui Fan, Idit Keidar, Ofer Kiselov, and Dmitri Perelman. 2011. Café: Scalable task pools with adjustable fairness and contention. In *DISC*.
[2] Nachshon Cohen and Erez Petrank. 2015. Efficient memory management for lock-free data structures with optimistic access. In *SPAA*.
[3] Robert Colvin and Lindsay Groves. 2005. Formal verification of an array-based nonblocking queue. In *ICECCS*.
[4] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A highly-efficient wait-free universal construction. In *SPAA*.
[5] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the combining synchronization technique. In *PPoPP*.
[6] John Giacomoni, Tipp Moseley, and Manish Vachharajani. 2008. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *PPoPP*.
[7] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. 2010. Cache-aware lock-free queues for multiple producers/-consumers and weak memory consistency. In *OPODIS*.
[8] James R. Goodman, Mary K. Vernon, and Philip J. Woest. 1989. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *ASPLOS*.
[9] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. 1983. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *TOPLAS* 5, 2 (1983).
[10] Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. 2013. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *CF*.
[11] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*.
[12] Maurice Herlihy. 1991. Wait-free synchronization. *TOPLAS* 13, 1 (1991).
[13] Maurice Herlihy, Beng-Hong Lim, and Nir Shavit. 1995. Scalable concurrent counting. *TOCS* 13, 4 (1995).
[14] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *TOPLAS* 12, 3 (1990).
[15] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. The baskets queue. *OPODIS* (2007).
[16] Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. 2013. Fast and scalable, lock-free k-FIFO queues. In *PaCT*.
[17] Alex Kogan and Maurice Herlihy. 2014. The future(s) of shared data structures. In *PODC*.
[18] Alex Kogan and Yossi Lev. 2017. Transactional lock elision meets combining. In *PODC*.
[19] Edya Ladan-Mozes and Nir Shavit. 2008. An optimistic approach to lock-free FIFO queues. *Distributed Computing* 20, 5 (2008).
[20] Doug Lea. 2009. The java concurrency package (JSR-166).
[21] Maged M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *TPDS* 15, 6 (2004).
[22] Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*.
[23] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. 2005. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA*.
[24] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *PPoPP*.
[25] Niloufar Shafiei. 2009. Non-blocking array-based algorithms for stacks and queues. In *ICDCN*.

[26] Philippas Tsigas and Yi Zhang. 2001. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *SPAA*.
[27] Chaoran Yang and John Mellor-Crummey. 2016. A wait-free queue as fast as fetch-and-add. In *PPoPP*.