# BQ: A Lock-Free Queue with Batching[*]

Gal Milman
Technion, Israel
galy@cs.technion.ac.il

Alex Kogan
Oracle Labs, USA
alex.kogan@oracle.com

Yossi Lev
Oracle Labs, USA
levyossi@icloud.com

Victor Luchangco
Oracle Labs, USA
vluchangco@gmail.com

Erez Petrank
Technion, Israel
erez@cs.technion.ac.il

## ABSTRACT

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 batch lengths), compared to previous queue implementations.

## KEYWORDS

Concurrent Algorithms; Concurrent Data Structures; Lock-Freedom; Linearizability; FIFO Queue

**ACM Reference Format:**
Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. BQ: A Lock-Free Queue with Batching. In *SPAA '18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, July 16–18, 2018, Vienna, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3210377.3210388

## 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.

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

*SPAA '18, July 16–18, 2018, Vienna, Austria*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5799-9/18/07…$15.00
https://doi.org/10.1145/3210377.3210388

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 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$^{\text{TM}}$ 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 algorithm's correctness is laid out briefly in Section 7. Section 8 describes measurement results. Further results, lock-freedom arguments, supplementary implementation details and the memory management mechanism are deferred to [23] (the full version of this paper).

## 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 promoted 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. [27] 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, 26]. Moir et al. [24] 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. [25] 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. [28] 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 at most $\#enqueues$, 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 $\ell^{th}$ 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 $\ell^{th}$ 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, in Subsection 6.2 we describe the algorithm, and in Subsection 6.3 we refer to memory management.

### 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

**Table 1: Underlying Data Structures**

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

and Count or Announcement object (*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 [23]. Measurements demonstrate that this variation 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. 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 commit 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 commit 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**

```
EnqueueToShared(item)
   newNode = new Node(item, NULL)
   while (true)
      tailAndCnt = SQTail
      if (CAS(&tailAndCnt.node->next, NULL, newNode))
         // newNode is linked to the tail
         CAS(&SQTail, tailAndCnt, ⟨newNode, tailAndCnt.cnt
              + 1⟩)
         break
      head = SQHead
      if head consists of Ann: // (head.tag & 1 != 0)
         ExecuteAnn(head.ann)
      else
         CAS(&SQTail, tailAndCnt, ⟨tailAndCnt.node->next,
              tail.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**

```
Item* DequeueFromShared()
   while (true)
      headAndCnt = HelpAnnAndGetHead()
      headNextNode = headAndCnt.node->next
      if (headNextNode == NULL)
         return NULL
      if (CAS(&SQHead, headAndCnt, ⟨headNextNode,
           headAndCnt.cnt + 1⟩))
         return headNextNode->item
```

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

**Listing 3: HelpAnnAndGetHead**

```
PtrCnt HelpAnnAndGetHead()
   while (true)
      head = SQHead
      if head consists of PtrCnt: // (head.tag & 1 == 0)
         return head.ptrCnt
      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.

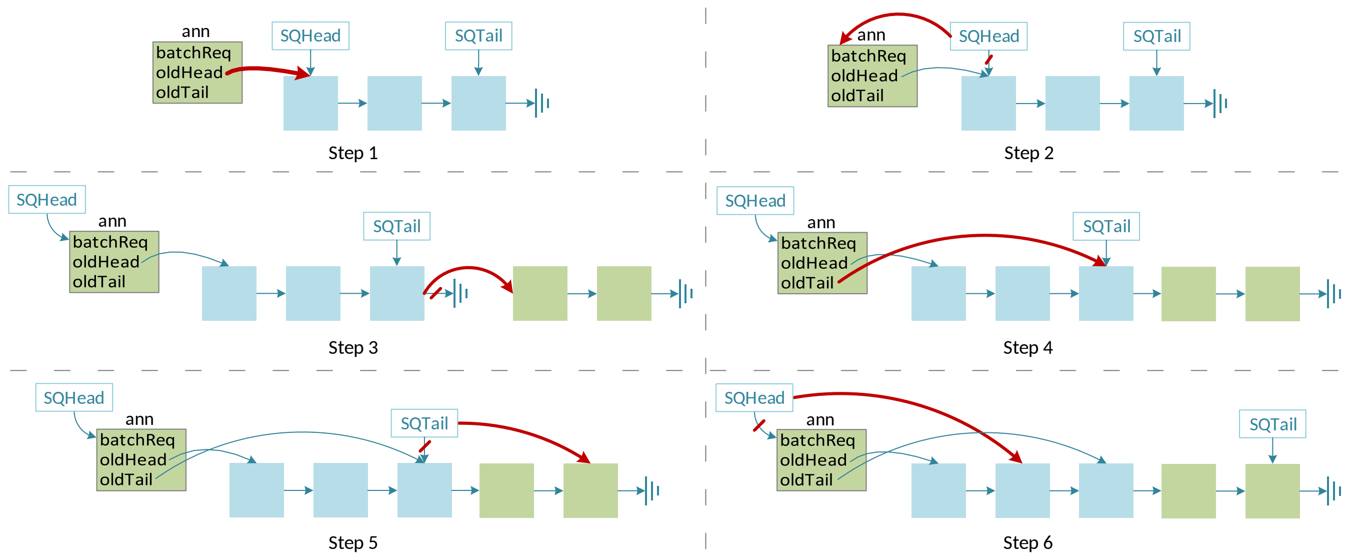

**Figure 1: Steps of the Batch Execution**

(1) Setting *ann->oldHead* to the head of the queue right before committing 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) Promoting *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).

**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 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. We will describe them here, but present their pseudo-code (which is straight forward) in [23]. Thereafter, 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* 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 committing all preceding pending operations, as well as committing the current operation. The implementation of *Dequeue* is similar. If *Dequeue* succeeds, it returns the dequeued item, otherwise (the queue is empty when the operation takes effect) it returns NULL.

*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. In addition, *FutureEnqueue* updates the numbers of pending enqueue operations. *FutureDequeue* operates similarly, but updates the numbers of pending dequeue operations and excess dequeues. *FutureEnqueue* also 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*.

*Evaluate* receives a future and ensures it is applied when the method returns. 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 by calling *ExecuteBatch*. After the batch operation's execution completes, while new operations may be applied to the shared queue by other threads, *PairFuturesWithResults* (described next) is called to pair the operation results to the appropriate futures of operations in *opsQueue*.

*ExecuteBatch* and then *PairFuturesWithResults* are called only if *opsQueue* consists of at least one enqueue operation. If all pending operations are dequeues, we pursue a different course of action, on which we elaborate in Subsection 6.2.3.

*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 6: PairFuturesWithResults**

```
73 PairFuturesWithResults(oldHeadNode)
74    nextEnqNode = threadData.enqsHead
75    currentHead = oldHeadNode
76    noMoreSuccessfulDeqs = false
77    while threadData.opsQueue is not empty:
78       op = threadData.opsQueue.Dequeue()
79       if (op.type == ENQ)
80          nextEnqNode = nextEnqNode->next
81       else // op.type == DEQ
82          if (noMoreSuccessfulDeqs ||
83             currentHead->next == nextEnqNode)
84             // The queue is currently empty
85             op.future->result = NULL
86          else
87             currentHead = currentHead->next
88             if (currentHead == threadData.enqsTail)
89                noMoreSuccessfulDeqs = true
90             op.future->result = currentHead->item
91       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 94). 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 97. If there is at least one successful dequeue, the dequeues take effect at once using a single CAS operation in Line 105. 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 7: ExecuteDeqsBatch**

```
92  ⟨unsigned int, Node*⟩ ExecuteDeqsBatch()
93     while (true)
94        oldHeadAndCnt = HelpAnnAndGetHead()
95        newHeadNode = oldHeadAndCnt.node
96        successfullDeqsNum = 0
97        repeat threadData.deqsNum times:
98           headNextNode = newHeadNode->next
99           if (headNextNode == NULL)
100             break
101          ++successfullDeqsNum
102          newHeadNode = headNextNode
103       if (successfullDeqsNum == 0)
104          break
105       if (CAS(&SQHead, oldHeadAndCnt, ⟨newHeadNode,
              oldHeadAndCnt.cnt + successfullDeqsNum⟩))
106          break
107    return ⟨successfullDeqsNum, oldHeadAndCnt.node⟩
```

**Listing 8: PairDeqFuturesWithResults**

```
108 PairDeqFuturesWithResults(oldHeadNode, successfulDeqsNum)
109    currentHead = oldHeadNode
110    repeat successfulDeqsNum times:
111       currentHead = currentHead->next
112       op = threadData.opsQueue.Dequeue()
113       op.future->result = currentHead->item
114       op.future->isDone = true
115    repeat threadData.deqsNum - successfulDeqsNum times:
116       op = threadData.opsQueue.Dequeue()
117       op.future->result = NULL
118       op.future->isDone = true
```

### 6.3 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. Due to space limitations, all related details appear in [23]. All measurements include use of memory reclamation.

## 7 CORRECTNESS

In this section we argue about the correctness of the algorithm. We start with an abstraction of the queue in Subsection 7.1, and then specify the linearization points in Subsection 7.2. We argue about lock-freedom in [23].

## 7.1 The Abstract State of the Queue

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 $SQHead$.
- 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 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.

## 7.2 Linearization Points

From the above definitions of the abstract head and the abstract state of the queue, we derive linearization points for all operation types. These linearization points are detailed next.

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`). All dequeues of a batch operation that contains only dequeue operations are linearized one after another, in the moment of reading the next pointer in Line 98 (of Method $ExecuteDeqsBatch$ in Listing 7) 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 99), or due to completing a traversal of $deqsNum$ items.

An enqueue operation takes effect when the next pointer of the last node in the list is modified from `NULL` to point to a new node. Similarly, 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. This modification may be carried out by a helping thread rather than the thread that invoked the operation.

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 described 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 described 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 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.

Each data point $[x, y]$ in the graphs in Figure 2 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.

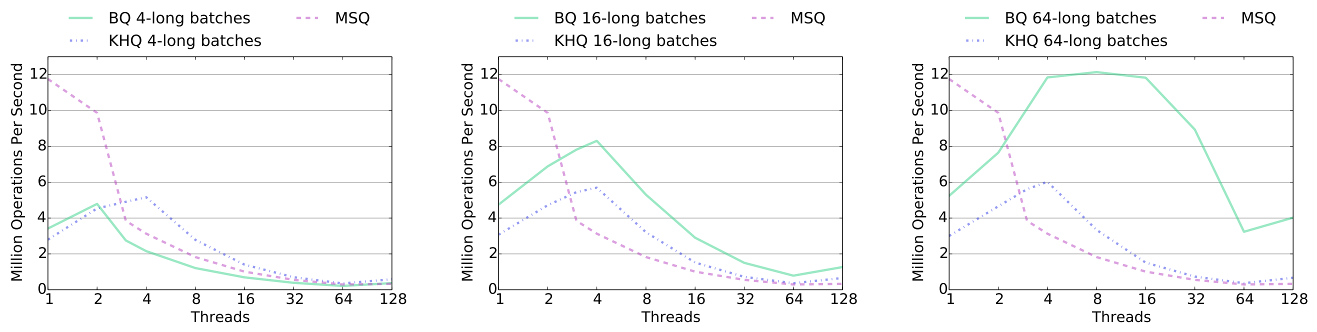

**Figure 2: Throughput for 4, 16 and 64 long batches respectively**

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 2. Additional measurements we conducted demonstrate further improvement for 256-long batches in comparison to 64-long batches, especially for executions with many threads.

## 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] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. BQ: a lock-free queue with batching (full paper). (2018). http://www.cs.technion.ac.il/~erez/Papers/bq-full.pdf
[24] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. 2005. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA*.
[25] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *PPoPP*.
[26] Niloufar Shafiei. 2009. Non-blocking array-based algorithms for stacks and queues. In *ICDCN*.
[27] Philippas Tsigas and Yi Zhang. 2001. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *SPAA*.
[28] Chaoran Yang and John Mellor-Crummey. 2016. A wait-free queue as fast as fetch-and-add. In *PPoPP*.