Data Structure Research Paper

BQ: A Lock-Free Queue with Batching

John Mirschel and Timothy Rigby

March 2019

Abstract

This paper describes an implementation of a lock-free queue that utilizes batching to increase performance by up to 16x over standard lock-free queues. We attempted to implement the design and algorithms from the original research paper BQ: A Lock-Free Queue with Batching and tried to reproduce similar functionality and performance. The original implementation of this data structure was done in C++, but we wanted to try and build this in Java so we could compare the results of how the data structure performs in two different languages.

1 Introduction

Our implementation of the lock-free queue provides all of the functionality available to a sequential queue. Our lock free batching extension builds upon the simple concurrent lock free queue implemented by Michael and Scott by vastly improving on scalability of the concurrent lock free queue. A concurrent queue has two access points of high contention, the head of the queue from which nodes are dequeued and the tail of the queue into where new nodes are enqueued. This inherently limits throughput of the data structure as only one thread can enqueue and one thread can dequeue at a time. Our version of a lock-free queue uses the idea of the future programming construct to perform batching operations as first seen in the paper written by Kogan and Herlithy[1]. Batching means to just group a sequence of standard operations to just one single batch operation, which then applies them together to the shared data structure.

Kogan and Herlithy originally proposed that operations with the same type (enqueue or dequeue) get added to the shared queue all at once. This means they execute each subsequence of enqueues together by appending them in order to the tail of the queue, and each subsequence of dequeues is removed from the head of the list all at the same time. The problem with this implementation is that performance can degrade if the operations in the batch frequently switch between enqueue and dequeue, as their implementation only allows for a batch to be built from sequential and like operations. The algorithm we implemented improves on this idea by batching these operations locally. Local batching allows our algorithm to handle both enqueue and dequeue operations in any order and build the result before applying it to the shared queue. This means it applies the batch operation all at once to the shared queue to reduce contention between multiple threads. This helps solve the problem that concurrent queues have with the contention of the head and tail pointers by batching these operations locally, which reduces the number of access attempts to the shared queue and improves scalability.

2 Related Works

The Lock Free Queue with Batching designed by Gal Milman et al. [2] is an extension of the concurrent lock-free queue by Michael and Scott [3]. The Michael and Scott queue serves as the baseline for many developments on the concurrent FIFO queue, yet it is not scalable and only allows for one Enqueuer and one Dequeuer thread at a time. Moir et al. [4] presented a queue that increased scalability by allowing pairs of concurrent equeue and dequeue method calls to exchange values and eliminate themselves without having to access the shared queue under certain circumstances. This concept reduces contention upon the shared queue, a concept adopted and furthered by the Lock Free Queue with Batching as it locally pairs multiple enqueue and dequeue operations before accessing the shared queue. The implementation of the Lock Free Queue with Batching most directly rises from the work done by Kogan and Herlihy which allows for batching of operations of the same type to be executed as a subsequence at once on the shared queue. The Lock Free Queue with Batching improves on the performance of the concurrent queue implemented by Kogan and Herlihy by allowing for alternation between enqueue and dequeue operations within a batch such as would most often be encountered in the general use case of a queue.

3 Overview

The Lock Free Queue with Batching is a direct extension of the basic concurrent Lock Free Queue. Our first step in building this data structure is to begin with a well-performing lock free queue implementation. The Lock Free Queue with Batching allows for quick computation of its size by maintaining counters with the head and tail of the shared queue. Batching threads will receive enqueue and dequeue operations to execute locally, building a subsequence of a queue that can be then applied to the shared queue atomically to perform multiple enqueue and dequeue operations in a single access to the shared queue. Local threads will also track the number of dequeue operations taking place to be able to quickly modify the shared queue as needed while applying the batch. When a batch is applied to the shared queue, a descriptor like object replaces the head of the queue in order to have all other threads assist the completion of the batching event.

3.1 Our Implementation

There are a few differences between our implementation and the papers original implementation. The first thing was that since the original paper implemented the data structure in C++ it made use of the *Union* operator which allowed different objects to share the same memory location. Java unfortunately does not have similar functionality due to not having low level memory access. To work around this we had to make some new data structures which we mention in Section 4.1.

Another key difference was that certain functionality regarding keeping track of excess dequeues as well as certain interface functions were described in the paper but had no examples in pseudocode to reference. Because of this, we had to write our own implementation of these methods which resulted in a slightly different layout than in the original version. This was one of the tougher obstacles we faced in our re-implementation since it require a lot of debugging, and trial and error to get the correct working version of the data structure from inference.

Our implementation of the Batching Queue is linearizable and lock-free. We guarantee progress across our threads due to the lock-free structure. With no critical section ever inaccessible by a thread, at least one thread is always guaranteed to have its operations succeed at any given time. The use of the atomic CompareAndSwap(CAS) operation ensures correctness through our

program. Each critical section and attempt to modify the shared queue is gated by infinite loops that will only exit upon the execution of a successful CAS operation in a linearizable fashion. The logic applied prevents any improper allocation of resources to disrupt the linkage of the queue. Each operation executes in the order it was called. Java's Atomics library was used for synchronization across threads. We make use of AtomicReferences to label the head and tail of our shared queue, and each node within the queue contains an AtomicReference to the next node in its sequeuence. The use of atomics in conjunction with the atomic CAS operation allow for all threads to synchronize their operations and not disrupt each other.

4 Algorithm Details

The Batching Queue extension to the Lock Free Queue allows for deferred operations to operate in batches of operations rather than having each enqueue and dequeue execute on the shared queue as they are called. The deferring of these operations allows for the user to aggregate pending operations within each thread to be executed at a later time. When a future method is called, either enqueue or dequeue, a Future object is returned to the caller who may evaluate it later. The execution of operations on the shared queue is delayed until the user explicitly evaluates the future of a deferred operation or a standard queue operation is called. At the point of evaluation, the pending operations of a thread are gathered to a single batched operation to be applied to the shared queue. After which the batch execution is finalized by locally filling in the return values of the pending operations. This locally batched execution and value filling reduces the overhead of having multiple threads compete for alteration of the shared queue.

4.1 Data Structures

```
public class Node<T>  {T val; AtomicReference<Node<T>> next;}

public class Future<T> {T returnVal; boolean isDone;}

public class NodeWithCount<T> {Node<T> node; int count;}
```

```
public class FutureOp<T> {boolean isEnqueue; Future<T> future;}

public class NodeCountOrAnn {
    boolean isAnnouncement;
    NodeWithCount nodeWCount;
    Announcement announcement;
}


public class BatchRequest<T>{
    Node<T> firstEnq;
    Node<T> lastEnq;
    int numEnqs;
    int numDeqs;
    int numExcessDeqs;
}

public class NodeCountOrAnn {
    boolean isAnnouncement;
    NodeWithCount nodeWCount;
    Announcement announcement;
}

public class ThreadData<T> {
    Queue<FutureOp> opsQueue;
    Node<T> enqHead;
    Node<T> enqTail;
    int numEnqs;
    int numDeqs;
    int numExcessDeqs;
}
```

---

1. **Node Class:** The Node class is a standard node class for a linked list implementation.

2. **Future Class:** The Future class is going to be containing a *result* which will be holding the return value of the deferred operation that generated the Future, and a isDone value so we know if the deferred operation has been completed. This *result* will only hold relevant data after the operation has been executed.

3. **NodeWithCount Class:** The NodeWithCount class is a standard node class but with a integer counter attached to perform double CAS operations.

4. **FutureOp Class:** The FutureOp class is used to hold a deferred operation such as an enqueue or a dequeue. It will also hold a copy of a Future to be used for evaluation.

5. **BatchRequest Class:** The BatchRequest class will be prepared by a thread that needs to initiate a batch, and it will hold the details of the batches operations. The fields *firstEnq* and *lastEnq* are references to the first and last nodes of the pending items to be put in to the shared queue. While the fields *numEnqs*, *numDeqs*, and *numExcessDeqs* are details about the batch.

6. **NodeCountOrAnn Class:** The NodeCountOrAnn is used to hold either an Annoouncement or a NodeWithCount. The class will have a boolean value *isAnnouncement*. If this is set to true, then the shared head holds an *announcement* object. If it is false then the shared head holds an *nodeWCount* object.

7. **ThreadData Class:** This is the class that will be used to store all the local data for a given thread. The *opsQueue* is a standard non thread-safe Queue of FutureOps that is used to store all operations received by a thread. The *enqHead* and *enqTail* fields are used to access and keep track of the queue of FutureOps. The last three fields *numEnqs*, *numDeqs*, and *numExcessDeqs*are used to store the prepossessing data so we are able to manage where the head and tail should be pointing after applying a batch.

4.2 Algorithm Implementation

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. It's caller can be either the batch's initiating thread or a helping thread that encountered the announcement while trying to complete its own operation.

EnqueueToShared: This method appends an item after the tail of the shared queue using two CAS operations. It first updates the shared tail's next node to point to the new node and item, and then updates the shared tail to point to the new node and item. If another thread obstructs operation by enqueueing its own item concurrently, *EnqueueToShared* will try and help complete the obstructing operation before retrying its own operation. This is possible through the use of two CAS operations to link the node and update the shared tail. This method can be obstructed by either a singular operation or a batch operation.

DequeueFromShared: If the queue is not empty when the dequeue operation takes effect, this method extracts an item from the head of the shared queue and returns it. Otherwise, it returns NULL if taking effect on an empty queue. This method will also help assist ongoing batch operations to complete first through its calling of *HelpAnnAndGetHead* in its execution.

HelpAnnAndGetHead: This auxiliary method helps announcements to complete their execution by returning the shared head's *NodeCountOrAnnouncement*. Threads will then be able to check for the presence of an *Announcement* object and assist completion if present. With no *Announcement* to operate on the thread will be able to execute on the shared head's *NodeWithCount* reference.

ExecuteBatch: This method is responsible for executing the batch. It receives a *BatchRequest* object and creates a new announcement for it. Before storing this announcement in the head it checks to see if the head contains an announcement. If the head already contains an announcement, it helps it to complete its execution. Otherwise, this method will replace the

head of the shared queue with this new announcement object, encouraging all other threads to help it complete.

ExecuteAnnouncement: is called by *ExecuteBatch* after installing an Announcement object in the shared head or by other threads that otherwise encounter an Announcement object in the shared head. *ExecuteAnnouncement* will carry out an Announcement's batch. If any of the steps in the execution of the batch have been completed by a competing thread, the method moves on to the next step without taking effect on the shared queue.

The first step of *ExecuteAnnouncement* is to make sure that the items in the Announcement is linked to the shared queue, and that the old tail to which they were appended has been recorded in the Announcement object. If the items have already been appended to the queue and the old tail has been recorded in the Announcement it is clear that another thread has completed the linking, and the method breaks out of the linkage loop. Otherwise, we try to link the items to the queue though a compare and set operation on the next pointer of the current tail node of the shared queue. We then check whether the compare and set operated succeeded in linking the items. If the items were successfully linked, the old tail reference is stored to signify the successful linkage. Otherwise, we would try and help any other obstructing operations and restart the attempt at the linkage

The next step in the method is to update the reference of the shared queue's tail to reflect the newly appended nodes. The last step is to call updateHead so we can update the head of the shared queue to point to the last node that was dequeued by the batch. By doing so, this will also uninstall the announcement that was inserted in to the shared queues head and it will complete its handling.

UpdateHead: This methods main purpose is to update the head to the correct location after a batch operation is complete. The algorithm for this process is as follows: If the number of the batch's successful dequeues is at least the size of of the queue before applying the batch, then the head is determined by over *successfulDeqsNum* - *oldQueueSize* nodes, starting at the position of the node pointed to by old tail. Otherwise, we determine it by passing over *successfulDeqsNum* and by starting at the old dummy node. Finally, the head is moved to the correct position after the batch is applied.

Interface Methods: The following methods are ones that are exposed to the user. They consist of *Enqueue*, *Dequeue*, *FutureEnqueue*, *FutureDequeue* and *Evaluate*. The first method *Enqueue*, is for if the user is just trying to insert a single item in to the shared queue. By calling *Enqueue* the thread will just push one single item on to the shared queue. *Dequeue* works similar to the *Enqueue* method in that it will just attempt to dequeue one item from the shared queue. If the queue is empty then *Dequeue* will just return null. *FutureEnqueue* is for the user to create a *FutureOp* object representing an enqueue operation to be inserted in to the *ThreadData* opsQueue. *FutureEnqueue* will also keep track of the number of pending enqueue operations so it knows the amount of enqueues in the batch. *FutureDequeue* operates in a similar way as *FutureEnqueue*, but is instead handling dequeue operations and keeping track of the number of pending dequeues. The final method is *Evaluate*, which receives a *Future* and makes sure that it is applied when the method returns. If the Future has already been applied from the outset, then the result is immediately returned. Otherwise, all the items in the local *ThreadData* queue *opsQueue* will be applied all at once by calling the method *ExecuteBatch*.

5 Results

The first performance results we'll talk about is comparing our lock-free implementation of BQ compared to the performance of a standard implementation of a lock-free queue. The goal here was to measure overall run-time on 5000 operations of random enqueue and dequeue operations while also varying the batch size amount and amount of threads. In *Figure 1* we see the smallest batch size of 4 perform about expected. With smaller batch sizes, BQ is not fully utilizing the performance increase of batching operations. With batch sizes this small, the standard lock-free queue slightly out performs BQ. The hardware used for analysis was an Intel Core i7-7700HQ CPU @ 2.80GHz with 16GB of RAM.

We start to see improvement over the standard lock-free implementation with a batch size of 16. This result was expected because as the batch sizes increase so should the performance of BQ as the benefit on contention due to the volume of batched operations outweighs the overhead of the batching operations. The last graph is with a batch size of 64, and is where we see the biggest performance increase. Increasing the load from this point forward

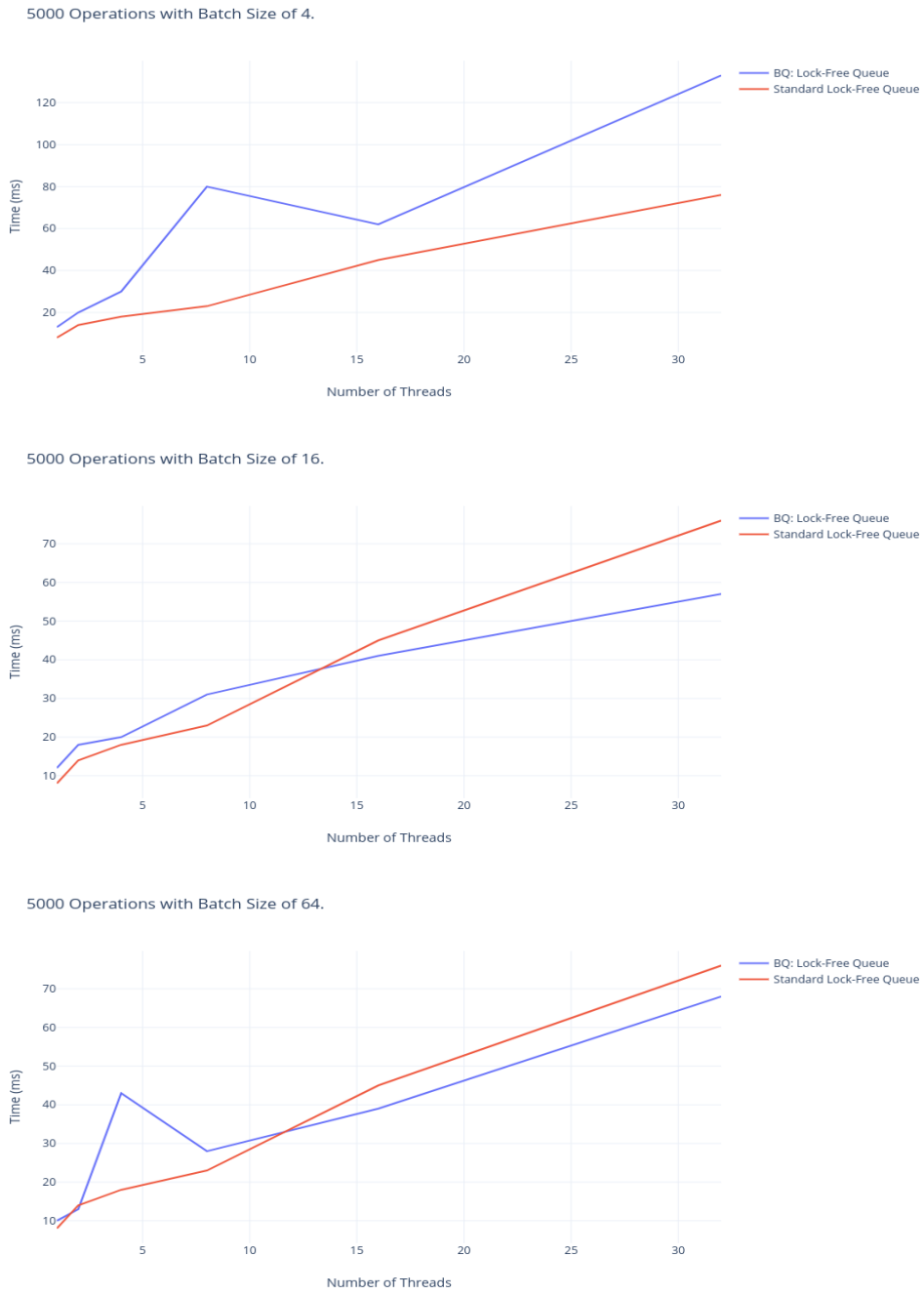


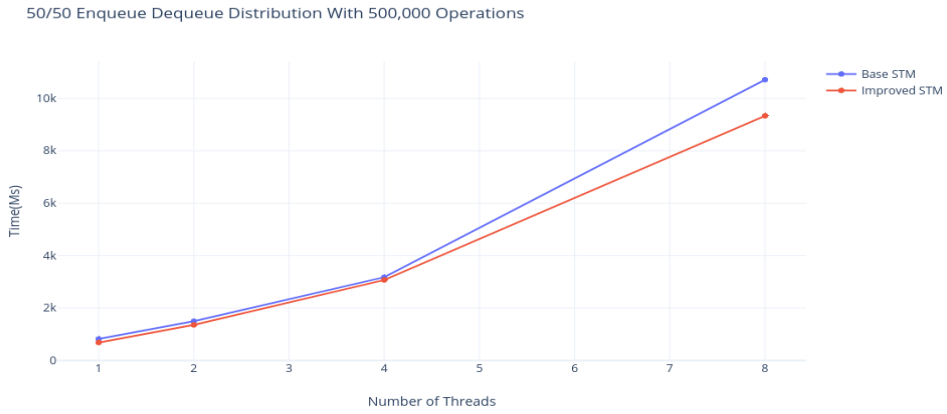
Figure 1: BQ vs Lock-Free Queue

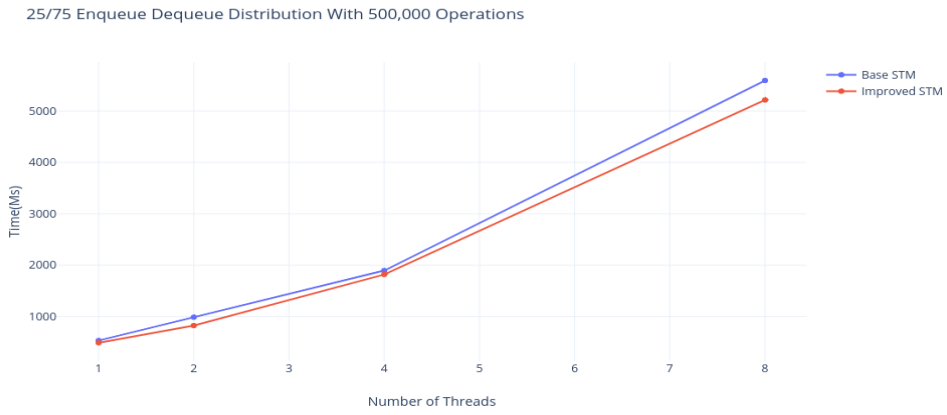
would further increase the performance of BQ compared to the lock-free queue.

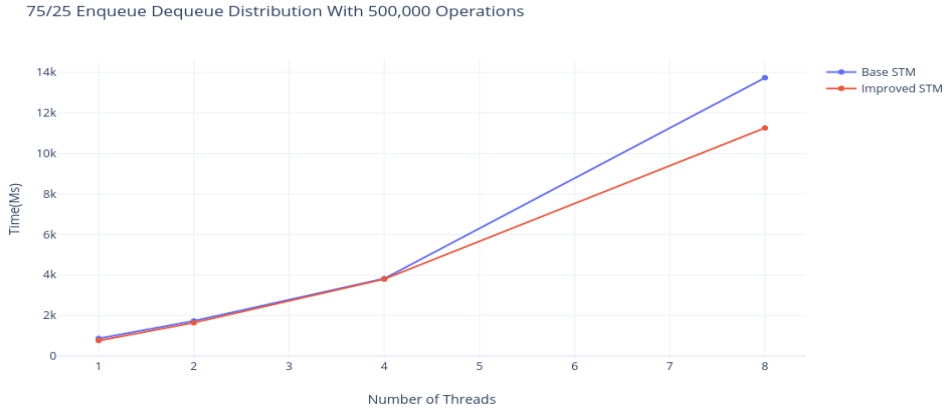
6 STM Implementation and Results

Another part of our project was creating an Software Transactional Memory (STM) [5] implementation of our lock-free data structure. Since our implementation was done in Java, we decided to go with the Java STM Deuce [6]. This STM library is older and lacks proper documentation, but makes up for it by being easy to implement. To create transactions in Deuce all you need to do is surround your methods that execute atomic operations with the @Atomic flag, and you can also add a parameter for a certain number of retries to catch any exceptions thrown by the method attempting to execute. This will the re enter the critical section that was previously marked with the @Atomic flag.

To analyze the performance of our STM implementation of our Batching Queue, we varied the distribution of operations taking effect on the queue (either enqueue or dequeue) as well as the number of threads competing to complete their execution. We created an initial STM implementation and then improved on this implementation to increase runtime while maintaining correctness. We did the testing for our two different STM implementations by performing 500,000 operations with 1,2,4,and 8 threads running. Through our testing we varied the distribution of operations from 50/50, 75/25, and 25/75 percent of enqueues to dequeues, respectively. Our results showed what was expected - our initial implementation utilized coarse-grained locking of the data structure which performed worse than the second implementation which utilized a more fine-grained locking approach. In our improvements to the STM implementation, we narrowed down the field of what was being locked down in order to have each thread hung up for less time in methods that didn't require atomicity.


50/50 Enqueue Dequeue Distribution With 500,000 Operations
Base STM
Improved STM
Time(Ms)
0
2k
4k
6k
8k
10k
1
2
3
4
5
6
7
8
Number of Threads


25/75 Enqueue Dequeue Distribution With 500,000 Operations
Base STM
Improved STM
Time(Ms)
1000
2000
3000
4000
5000
1
2
3
4
5
6
7
8
Number of Threads


75/25 Enqueue Dequeue Distribution With 500,000 Operations
Base STM
Improved STM
Time(Ms)
0
2k
4k
6k
8k
10k
12k
14k
1
2
3
4
5
6
7
8
Number of Threads

7 Conclusion

The Lock-Free Batching Queue outperforms the standard Lock-Free Queue by a measurable margin when used to its potential with large batch sizes. When using a low size of batched operations, such as four, the BQ's increased overhead of the batching methodology outweighs the benefit of the locally batched operations and does not out perform the standard Lock-Free Queue. Yet, as the batch sizes increase to 16 and further the BQ begins to show the benefit of locally batching and preprocessing Enqueue and Dequeue operations. The standard Lock-Free Queue experiences high contention and low scalability through its efforts for each thread to access the shared head and tail node, which the BQ's advantages shine when multiple threads are able to locally process large numbers of operations and minimize contention on the shared queue. Particularly as the number of threads in use increases, the standard queue is bogged down by the high contention on the shared object while the BQ is able to maintain throughput. The hardware used for analysis was an Intel Core i7-7700HQ CPU @ 2.80GHz with 16GB of RAM.

To reiterate, the Batching Queue's greatest advantage over other Lock-Free Queue implementations is the utilization of large sized batch operations being applied. Local processing of enqueues and counting of dequeues allows for each thread to hold the shared queue for the shortest amount of time possible to perform the operation. The disadvantage of the Batching Queue shows when applying small batch sizes as the increased computation required to perform the local preprocessing is most costly than continuously trying to compete for the shared queue in the way that the standard Lock-Free Queue does. As for potential improvements to the design of the Batching Queue, none stood out to us as notable improvements through our implementation. The research applied and summarized through the Batching Queue seems to be on the forefront of large operation concurrent Lock Free Queues and is highly performant in the situations it is designed to be.

References

[1] Alex Kogan and Maurice Herlihy. "The future(s) of shared data structures". In: *PODC* (2014).

[2] Gal Milman, Victor Luchangco, Alex Kogan, et al. "BQ: A Lock-Free Queue with Batching". In: *SPAA* (July 2018).

[3] Maged M. Michael and Michael L. Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *PODC* (1996).

[4] Mark Moir, Daniel Nussbaum, Ori Shalev, et al. "Using elimination to implement scalable and lock-free FIFO queues". In: *SPAA* (2005).

[5] Deli Zhang, Pierre Laborde, Lance Lebanof, et al. "Lock-Free Transactional Transformation for Linked Data Structures". In: *ACM* (June 2018).

[6] Guy Korland, Nir Shavit, and Pascal Felber. "Noninvasive concurrency with Java STM". In: *MultiProg* (2010).