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 do 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 thing was that certain functionality regarding keeping track of excess dequeues, and certain interface functions were described in the paper, but there was no examples in pseudo code to reference. Because of this, we had to write our own implementation of these methods which resulted in a bit different layout that 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.

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 just 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 now if the deferred operation has been completed.
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 we are using the *announcement*field, and if it false then we are using the *nodeWCount* field.
7. **ThreadData Class:** This is the class that will be used to store all the local data for a given thread. The *opsQueue* is just 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.

DequeueFromShare: 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 head *NodeWithCount* if there is no announcement in place or the Announcement object if there is an announcement in progress, allowing the calling thread to assist execution. This method will return the head of the shared queue, which is a *NodeCountOrAnnouncement* object.

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 $uccess\,f\,ulDeqsNum$ - *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.

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

6 STM Implementation and Results

Another part of our project was creating an STM [30] implementation of our lock-free data structure. Since our implementation was done in Java, we decided to go with the Java STM Deuce [16]. 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.

Recent advances in transactional data structure design can deliver a more fine-tuned implementation [27, 31, 18, 3, 19, 24], however this will require the re-engineering of the data structure design and is outside the scope of our current work. Other alternatives to the use of STM are the use of multi-resource locking [32] or a multi-word CAS algorithm [9]. We plan to explore the use of these techniques in our future work. Furthermore, in our future work we plan to explore the guarantee wait-free progress by using the facilities provided by the Tervel library [8].

Per the performance evaluation, we decided to vary the the number of threads and also vary the split. The performance evaluation follows the experimentation methodology for the analysis of non-blocking data structures presented by Izadpanah et. al [13].

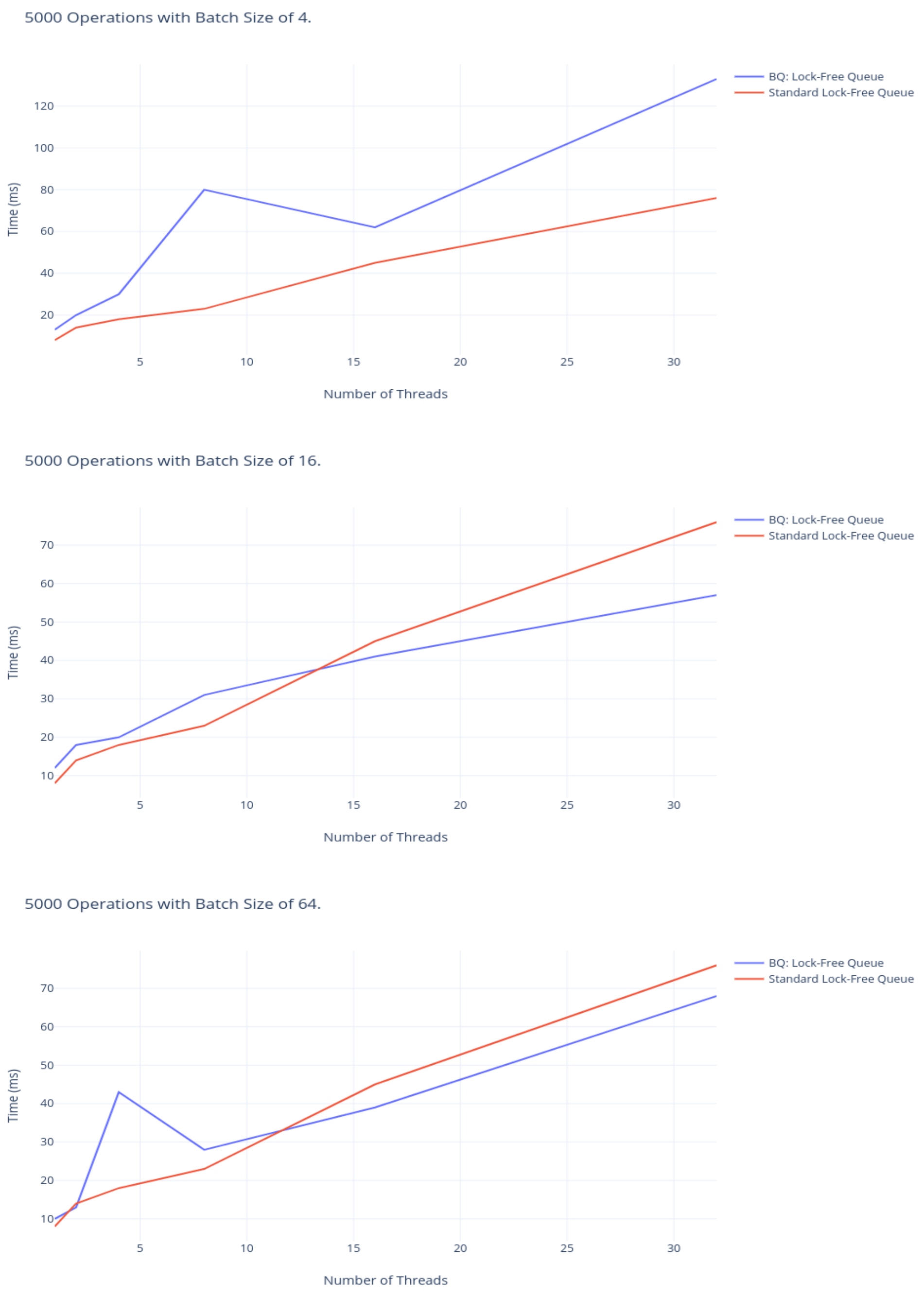

Figure 1: BQ vs Lock-Free Queue

References

[1] T.-H. Ahn, D. Dechev, H. Lin, H. Adalsteinsson, and C. Janssen. Evaluating performance optimizations of large-scale genomic sequence search applications using sst/macro. In *Proceedings of the 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, Noordwijkerhout, The Netherlands, July 2011.

[2] A. Barrington, S. Feldman, and D. Dechev. A scalable multi-producer multi-consumer wait-free ring buffer. In *Proceedings of the 30th ACM/SIGAPP Symposium on Applied Computing*, Salamanca, Spain, April 2015.

[3] V. Cook, Z. Painter, C. Peterson, and D. Dechev. Read-uncommitted transactions for smart contract performance. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, Dallas, TX, July 2019.

[4] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *Proceedings of 10th International Conference on Principles of Distributed Systems*, pages 142–156, Bordeaux, France, December 2006.

[5] D. Dechev, N. Rouquette, P. Pirkelbauer, and B. Stroustrup. *Programming and Validation Techniques for Reliable Goal-driven Autonomic Software*. Springer, Boston, MA, vasilakos a., parashar m., karnouskos s., pedrycz w. (eds) autonomic communication edition, 2009.

[6] D. Dechev and B. Stroustrup. Reliable and efficient concurrent synchronization for embedded real-time software. In *Proceedings of 3rd IEEE International Conference on Space Mission Challenges for Information Technology*, Pasadena, California, July 2009.

[7] D. Dechev and B. Stroustrup. Scalable nonblocking concurrent objects for mission critical code. In *Proceedings of 24th International Conference on Object-Oriented Programming Languages, and Applications*, Orlando, Florida, October 2009.

[8] S. Feldman, P. LaBorde, and D. Dechev. Tervel: A unification of descriptor-based techniques for non-blocking programming. In *Proceedings of the 15th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, July 2015.

[9] S. Feldman, P. LaBorde, and D. Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, 43(4):572–596, February 2015.

[10] S. Feldman, P. LaBorde, D. Dechev, and C. M. level Arrays. Wait-free extensible hash maps. In *Proceedings of the 13th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos, Greece, July 2013.

[11] S. Feldman, C. Valeraleon, and D. Dechev. An efficient wait-free vector. *IEEE Transactional on Parallel and Distributed Systems*, 27(3):654–667, May 2016.

[12] S. Feldman, D. Zhang, D. Dechev, and J. Brandt. Extending ldms to enable performance monitoring in multi-core applications. In *Proceedings of the Monitoring and Analysis for High Performance Computing Systems Plus Applications*, Chicago, IL, September 2015.

[13] R. Izadpanah, S. Feldman, and D. Dechev. A methodology for performance analysis of non-blocking algorithms using hardware and software metrics. In *Proceedings of the 19th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing*, York, UK, May 2016.

[14] R. Izadpanah, N. Naksinehaboon, J. Brandt, A. Gentile, and D. Dechev. Integrating low-latency analysis into hpc system monitoring. In *Proceedings of the 47th International Conference on Parallel Processing*, Eugene, OR, August 2018.

[15] A. Kogan and M. Herlihy. The future(s) of shared data structures. 2014.

[16] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with java stm. 2010.

[17] P. LaBorde, S. Feldman, and D. Dechev. A wait-free hash map. *International Journal of Parallel Programming*, 45:421–448, 2017.

[18] P. LaBorde, L. Lebanoff, C. Peterson, D. Zhang, and D. Dechev. Wait-free dynamic transactions for linked data structures. In *The 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, Washington DC, USA, February 2019.

[19] L. Lebanoff, C. Peterson, and D. Dechev. Check-wait-pounce: Increasing transactional data structure throughput by delaying transactions. In *Proceedings of the 19th International Conference on Distributed Applications and Interoperable Systems*, Lyngby, Denmark, June 2019.

[20] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. 1996.

[21] G. Milman, V. Luchangco, A. Kogan, E. Petrank, and Y. Lev. Bq: A lock-free queue with batching. 7 2018.

[22] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. 2005.

[23] Z. Painter, C. Peterson, and D. Dechev. Lock-free transactional adjacency list. In *Proceedings of the 30th International Workshop on Languages and Compilers for Parallel Computing*, College Station, TX, October 2017.

[24] C. Peterson and D. Dechev. A transactional correctness tool for abstract data types. *ACM Transactions on Architecture and Code Optimization*, 14(4), December 2017.

[25] M. Sottile, J. Dagit, D. Zhang, G. Hendry, and D. Dechev. Static analysis techniques for semi-automatic synthesis of message passing software skeletons. *ACM Transactions on Modeling and Computer Simulation*, 26(4), September 2015.

[26] M. Sottile, A. Dakshinamurhty, G. Hendry, and D. Dechev. Semi-automatic extraction of software skeletons for benchmarking large-scale parallel applications. In C. Montreal, editor, *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, May 2013.

[27] D. Zhang and D. Dechev. An efficient lock-free logarithmic search data structure based on multi-dimensional list. In J. Nara, editor, *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems*, June 2016.

[28] D. Zhang and D. Dechev. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactional on Parallel and Distributed Systems*, 27(3):613–626, April 2016.

[29] D. Zhang, G. Hendry, and D. Dechev. Tools for enabling automatic validation of large-scale parallel application simulations. In *Proceedings of 30th International Conference on Software Maintenance and Evolution*, Victoria, British Columbia, Canada, September 2014.

[30] D. Zhang, P. Laborde, L. Lebanof, and D. Dechev. Lock-free transactional transformation for linked data structures. 6 2018.

[31] D. Zhang, P. LaBorde, L. Lebanoff, and D. Dechev. Lock-free transactional transformation. *ACM Transactions on Parallel Computing*, 5(1), June 2018.

[32] D. Zhang, B. Lynch, and D. Dechev. Queue-based and adaptive lock algorithms for scalable resource allocation on shared-memory multiprocessors. *International Journal of Parallel Programming*, 43(5):721–751, August 2015.