Concurrent Queues with Relaxed Semantics

Calvo, Marlon Mixon, Jace Li, Weining
Epes, Chandler

April 2, 2020

Group 12

Contents

1 Quantifiability

The paper (Cook et al.) proposes a new conccurent correctness condition called Quantifiability. Instead of the more traditional approach of proving equivalence to a sequential history, Quantifiability looks at the outcome of the method calls. This approach is similar to the Serialization correctness condition (Papadimitriou), but Quantifiability methods must not return null during invalid states and instead be set to "pending" and then return a value later. For this strategy to work the method calls must be atomic and Quantifiabilty is also compositional. In designing Quantifiability they aimed for it to have the following properties:

- Conservation - What happens is what the methods did.
- Measurable - Method calls have a certain and measurable impact on the system state, not sometimes null.
- Compositional - Demonstratably correct objects and their methods may be combined into demonstratably correct systems.
- Unconstrained by Timing - Correctness based on timing limits opportunities for performance gains and incurs. verification overhead when comparing the method call invocation and response times to determine which method occurs first in the history.
- Lock-free, Wait-free, Deadlock-free, Starvation-free - Design of the correctness condition should not limit or prevent system progress.

Quantifiability embodies two principles that must be followed for it to be true:

1. Method Conservation - Method calls are first class objects in the system that must succeed, remain pending, or be explicity cancelled.
2. Method Quantifiability - Method calls have a measurable impact on the system state.

The paper provides the following formal definition of Quantifiability:

- Let $\vec{P}$ be the vector obtianed by applying vector addition to the set of vectors for the producer set of history H. Let $\vec{W}_{prod}$ be the vector obtained by applying vector addtion to the set of vectors $\vec{V}_{iprod}$ for the writer set of history H. Let $\vec{W}_{cons}$ be the vector obtained by applying vector addition to th set of vectors $\vec{V}_{icons}$ for the writer ste of history H. Let $\vec{R}$ be the vecotr obtained by applying vector addition to the set of vectors for the reader set of history H. Let $\vec{C}$ be the vector obtained by applying vector addition to the set of vectors for the consumer set of history H. Let $\vec{H}$ be a vector with each element initalized to 0. For each element i, if

  $$(\vec{P}[i] + \vec{W}_{prod}[i]) \geq 1)$$

  then

  $$\vec{H}[i] = \vec{P}[i] + \vec{W}_{prod}[i] + \vec{R}[i] + \vec{W}_{cons}[i] + \vec{C}[i]$$

  else

  $$\vec{H}[i] = \vec{P}[i] + \vec{W}_{prod}[i] + \vec{R}[i] + \vec{W}_{cons}[i] + \vec{C}[i]$$

  The history H is quantifiable if for each element i, $\vec{H}[i] \geq 0$.

A large advantage of defining correctness as properties over a set of vectors is that the size of a set of vectors grows at the rate of $O(n)$ with respect to n method calls in a history.

2 Quantifiable Queue

Quantifiable Queue is a queue structured as a doubly-linked tree of nodes designed around the quantifability properties described above. It allows two concurrent *enqueue* method calls to append their nodes and form a fork in the tree. *Enqueue* and *dequeue* methods are allowed to insert and remove nodes at any leaf nodes. It has the following methods / classes:

- Insert - A generalized operation to conserve unsatisfied dequeues. "If a *dequeue* operations is called on an empty [queue], we instead begin a [queue] of waiting [*dequeue*] operations by calling *insert* and designating the inserted node as an unfufilled [*dequeue*] operation" (Cook et al.).

- Remove - A generalized operation to conserve unsatisfied dequeues. Similar to *insert*, "if we call [*enqueue*] on a [queue] that contains unsatisfied [dequeues], we instead use *remove* to eliminate an unsatisfied [*dequeue*] operation, which then finally returns the value provided by the incoming [*enqueue*]" (Cook et al.).

- Enqueue - Wraps the *insert* and *remove* algorithms. Add node to a random index in the tail array to avoid contention. If the node has an unsatisfied *dequeue* operation, then it will be fulfilled, else a node will be inserted. A thread failing to make progress will result in a fork in the queue to increase the chance of future *insert* success.

- Dequeue - Wraps the *insert* and *remove* algorithms. Similar to *enqueue*, it will retrieve a head node at a random index. If the node's operation is *enqueue*, the node is removed, else an unsatisfied *dequeue* operation is inserted in the queue.

- Node - Contains the stored value, designated operation, array of references to its children, a reference to its parent, and a reference to a descriptor.

- Desc - Contains the stored value, designated operation, and an active field that designates whether the operation is pending or completed. Can be used by an arbitrary thread to carry out the intended operation.

2.1 MRLock Algorithm and Implementation

We implemented QQueue in C++ using MRLock (Zhang, Lynch, and Dechev) such that the queue has a singular lock that the thread must obtain

to make any changes. The implementation has the following methods / classes:

- insert() - Similar to standard implementation.
- remove() - Similar to standard implementation.
- push() - Lock the queue using MRLock, if the tail of the queue is a pop operation, remove it from the queue and satisfy that operation, otherwise insert a new node into the queue, then unlock the queue using MRLock.
- pop() - Lock the queue using MRLock, if the queue is empty, insert a pop request into the queue to be satsified by the next push, otherwise remove the front-most value of the queue, then unlock the queue using MRLock.
- Node - Similar to standard implementation.
- Desc - Similar to standard implementation.

The algorithm holds the same correctness of the original QStack implementation. Due to the use of locks, we shifted from a wait-free implementation, to a starvation-free algorithm, since once a thread grabs the lock, they will eventually be able to queue/deque. Our MRLOCK QQueue maintains the definitions of quantifiabilty as method calls are always perseved, stored in the queue for later use, and allows each method to define the impact on their system state. Although not specifically implemented, appending scalar values for each method call is straightforward, allowing validation through the verification algorithm (Cook et al.).

2.2 QQueue Algorithm and Implementation

For our QQueue implementation, we followed the QStack pseudocode from (Cook et al.) closely. Due to the nature of a queue, we had to support

dequeing from the head, and pushing to the tail (where a stack only pushes / pops from the tail). To allow such, we modified the *remove*() function, to allow deletion on non-leaf nodes. In general our algorithm functioned as such:

- queue() - We create a Node to insert to the Queue. If we have any unfinished Deque's we try to resolve those, and if not, we simply enqueue new item. To resolve a deque, we call the remove() operation on a head node if the head node is also a unfinished Deque we remove it, else, we just retry queue(). In any other case, we simply try to add a queue and break if succesful.

- deque() - Similar to the style of queue(), we try to remove a valid operation (Queue in Head node). If we reach an invalid state (such as empty queue or Deque Node in head), we insert a new invalid deque call to the queue. In our implementation, we had to be careful with checking if the Node was due to an empty queue, or invalid index in the head array. As such meant two different cases (one we re-try deque(), while another we insert a Node because of an invalid state).

- insert() - Very similar to the original insert() operation, since we utilize insert only at the tail (just like QStack). We opted to ignore the top global variable, and utilized continous tail / head updating to remove old traces of deleted / superseded nodes.

- remove() - To allow removing from the head(), we had to be able to do these things: delete cur node from head, add cur's children to head, and update tail if necessary. Like insert, once we have control of the Node, we try to check if the Node is in the right index in head. If so, we grab all of the Node's children and add it to head, while updating their prev. This can be done with a thread-safe list. Furthermore, we remove cur from head and tail (if necessary).

2.3 Correctness and Progress Conditions

Similar to the QStack, we allow wait-freedom with forkRequest. The number of operations that a thread performs in either a queue() or deque() operation is bounded by the FAIL THRESHOLD parameter, because insert() and remove() are non-waiting operations, and since we only retry FAIL THRESHOLD times, we are forced to stop at some time. Correctness condition is based on Quantifiability, similar to the QStack. Since during any queue() or deque(), we preserve the call (complete invalid Deque operations, insert failed Deque operations), and verification can be performed by traversing the tree for succesfull / unsuccesful operations, this implementation is Quantifiable. It also holds the k-FIFO feature, with k being infinite (like the QStack), since contention of threads increase the number of forks / leaf nodes. Although not necessarily part of quantifiability's correctness condition, it takes advantage of it to perform really well as the number of threads increase.

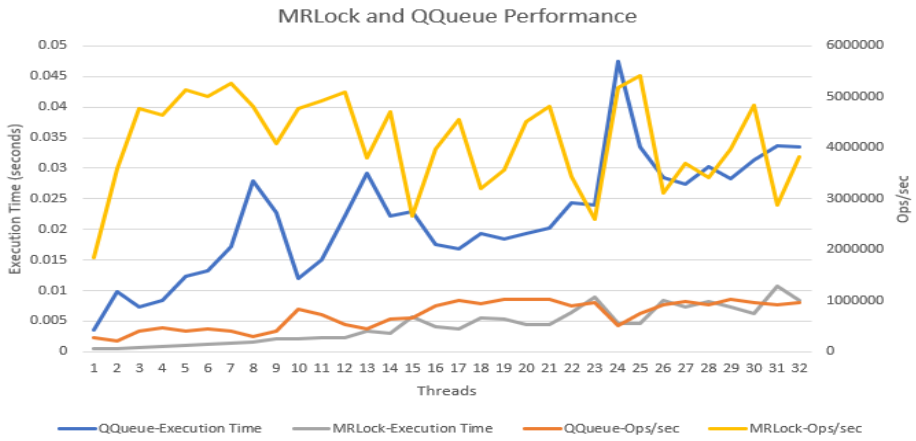

Figure 1: 60:40 enqueue:dequeue ratio

3 Quantifiable Queue vs MRLOCK QQueue

Shown in Figures 1, 2, and 3 are performance comparisons between our QQueue implemntation as described in the paper against the implementation using MRLOCK. We vary the thread count from 1 to 32 and record the execution time. We perform this test on a variety of different operation combinations to get a more thorough understanding of the performance differences.

As the number of threads increased, the throughput declined exponentially (Figure 1). This is mainly due to the nature of the locking algorithm, where the head and tail must be locked for queue/deque operations. Such causes every operation to be synchronized in a manner where only one operation occurs at a time. If head was not locked during a queue operation, or tail during a deque, performance will still slow down, but the coefficient of the decay may be approximately halved. This trend is evident for all queue/deque percentages, as regardless of the operation, it forces head and tail to be locked (Figure 1, Figure 2, Figure 3). The MRLock implementation has an advantage for single threaded performance, since it lacks the overhead of memory managment / list managment, as well as is inherently synchronous.

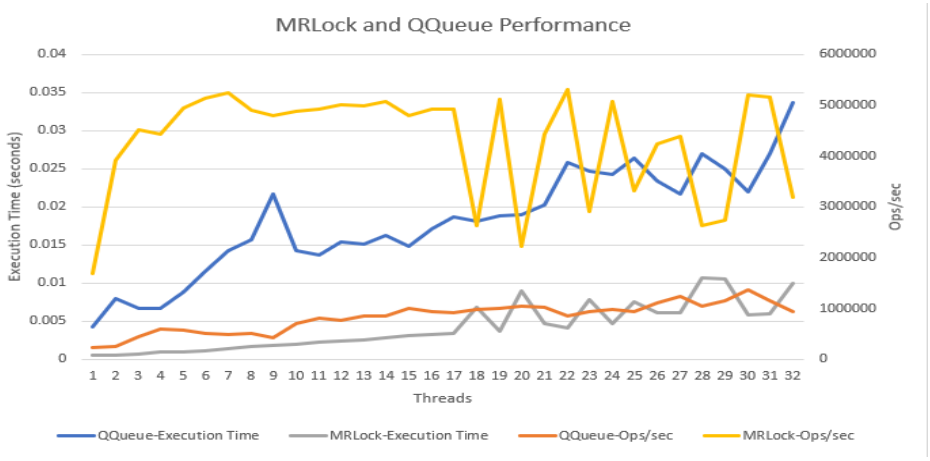

Figure 2: 75:25 enqueue:dequeue ratio

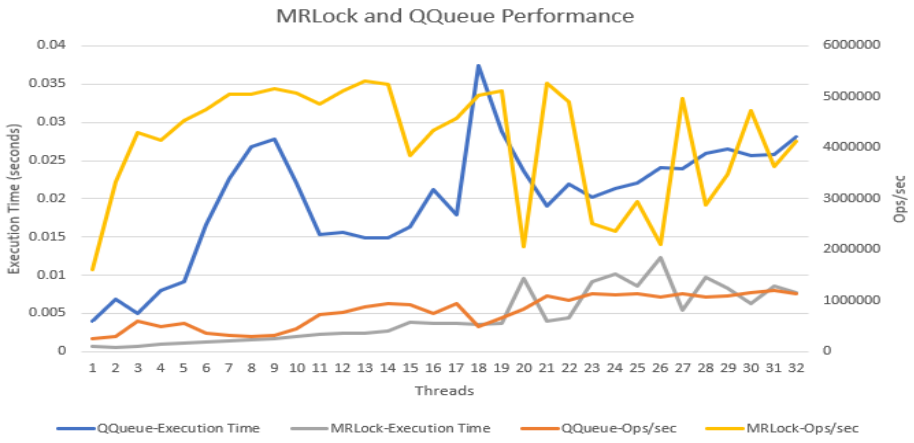

Figure 3: 80:20 enqueue:dequeue ratio

On the other hand, the QQueue implementation saw performance gains for increasing number of threads. This is mainly due to having multiple heads and tails. Furthermore, as the number of threads increases, the number of heads and tails increase, which scale up to reduce contention of threads significantly.

Another thing to note, was that there tests were performed on an i7-4790k with 4 cores / 8 threads, which one can see causes plateaus as the number of threads increase above 16. On more powerful systems, the QQueue implementation may scale better.

Works Cited

Cook, Victor, et al. "Quantifiability: Correctness Conditions from First Principles". 2019. Web. 12 Mar. 2020.

Papadimitriou, Christos. "The Serializability of Concurrent Database Updates". *Jounal of the ACM (JACM)*. 1979. Print.

Zhang, Deli, Brenda Lynch, and Damian Dechev. "Fast and Scalable Queue-Based Resource Allocation Lock on Shared-Memory Multiprocessors". *Principles of Distributed Systems: 17th International Conference*. 2013. Print.