Quasi-Linearizability: Relaxed Consistency For Improved Concurrency

Yehuda Afek, Guy Korland, and Eitan Yanovsky

Computer Science Department
Tel-Aviv University, Israel,
{afek|guykorla|eitanyan}@post.tau.ac.il

Abstract. Linearizability, the key correctness condition that most optimized concurrent object implementations comply with, imposes tight synchronization between the object concurrent operations. This tight synchronization usually comes with a performance and scalability price. Yet, these implementations are often employed in an environment where a more relaxed linearizability condition suffices, where strict linearizability is not a must.
Here we provide a quantitative definition of limited non-determinism, a notion we call Quasi Linearizability. Roughly speaking an implementation of an object is quasi linearizable if each run of the implementation is at a bounded "distance" away from some linear run of the object. However, as we show the limited distance has to be relative to some operations but not all.
Following the definition we provide examples of quasi concurrent implementations that out perform state of the art standard implementations due to the relaxed requirement. Finally we show that the Bitonic Counting Network non-deterministic behavior can be quantified using our Quasi Linearizable notion.

1 Introduction

Linearizability[5] is a useful and intuitive consistency correctness condition that is widely used to reason and prove common data structures implementations. Intuitively it requires each run to be equivalent in some sense to a serial run of the algorithm. This equivalence to some serial run imposes strong synchronization requirements that in many cases results in limited scalability and synchronization bottlenecks. In order to overcome this limitation, more relaxed consistency conditions have been introduced. Such alternative consistency conditions for concurrency programming include Sequential consistency[7], Quiescent consistency[6], Causal consistency[2], Release consistency[3], Eventual consistency[10] and Timed consistency[9]. But, the semantics of these relaxed conditions is less intuitive and the results are usually unexpected from a layman point of view. In this paper we offer a relaxed version of linearizability that preserves some of the intuition, provides a flexible way to control the level of relaxation and supports the implementation of more concurrent and scalable data structures.

For example, SEDA[11], the motivating and initiating reason for the current research is a common design pattern for highly concurrent servers, which heavily relies on thread pools. Such thread pools are composed from two elements (i) a set of threads ready to serve tasks and (ii) a task queue from which the threads consume their tasks. For the task queue, state of the art concurrent queue of Michael and Scott[8] is usually used. It is based on the fact that enqueue and dequeue may happen concurrently while threads trying to enqueue should race. Meaning such queue, which is not part of the server logic in a highly concurrent system, can become by itself a bottleneck limiting the overall SEDA system utilization. One can claim however, that more than often a thread pool does not need a strict FIFO queue, what is required is a queue with relaxed linearizability, i.e., that does not allow one task to starve, meaning bypassed by more than a certain number of tasks.

Another common pattern is the shared counter, which in many applications may become a bottleneck by itself. In order to trim down this contention point Aspnes et al.[6] offered a *counting network* which reduces the contention while maintaining a relaxed consistency condition called quiescent consistency. Such a relaxed counter can be used for example as an id generator, the output of this algorithm is a unique id for each requesting thread while a strict order is not required. This counter may also match other design patterns for example a "Statistical Counter". Modern servers expose many statistical counters, mainly for administration and monitoring. These counters count "online" every operation done on the server. Due to their run time nature these counters by themselves may easily become a contention point. However, sometimes there is no real need for accurate numbers but to capture the general trend. On the other hand the main drawback of the counting network algorithm is also its relaxed consistency, such relaxation does not provide any upper bound for the "inconsistency". We show in Section 6 that the upper bound is $N * W$ where N is the number of working threads, and W is the width of the counting network.

Two more common examples for widely used data structures are the Hash Table and the Stack. While there is a broad range of highly concurrent implementations for a Hash Table as with the former examples the need for a linearizable implementation is often too strict. A very common use case for a Hash Table is a Web Cache. In this case a cache miss while the data is in the Cache might not be a desirable behavior but can be sacrificed for a better scalability. More than that, even getting a stale data for a short while might not be a problem. A similar thing commonly happens with a Stack, a linearizable LIFO implementation can ordinarily be replaced with an almost LIFO implementation for a better scalability.

The above examples have motivated us to provide a quantitative definition of the limited non-determinism that the application requirements might allow. We define a consistency condition which is a relaxed linearizability condition with an upper bound on the non-determinism. Each operation must be linearizable at most at some bounded distance from its strict linearization point. For example, tasks may be dequeued from a queue not in strict FIFO order. That is, a task

t may be dequeued if no task that has been enqueued k tasks or more before t in a linearization order, has not yet been dequeued. Our definition is strong and flexible enough to define at the same time (continuing the above example) that a dequeue that returns empty may not be reordered, i.e., it has to be in its strict linearizable order. In this paper we introduce a formal definition of *quasi-linearizability* condition which captures this condition. This condition introduces some degree of non-determinism, but is useful to prove the quasi-linearizability of different implementations as exemplified in later sections.

1.1 Other relaxed consistency conditions

Many models were offered as weaker alternatives to Linearizability two of them are *Quiescent consistency*[6] and *Eventual consistency*[10].

Quiescent consistency provides high-performance at the expense of weaker constraints satisfied by the system. This property has two conditions:

1. Operations should appear in some sequential order (legal for each object).
2. Operations whose occurrence is separated by a quiescent state should appear in the order of their occurrence. An object is in a quiescent state if currently there is no pending or executing operation on that object.

Eventual consistency this is a specific form of weak consistency; e.g. a storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. The most popular system that implements eventual consistency is the Internet DNS (Domain Name System). Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the last update.

Both models, in most cases, allows better concurrency but on the other hand do not provide any strict upper bound or an adaptive way to determine the "inconsistency" gap when compared to Linearizability.

The contributions of this paper are, first a formal definition of *Quasi Linearizability*, second, two implementations of a FIFO queue that utilize this definition. Third, we empirically evaluate these implementations showing much better scalability relative to the state of the art implementations. Finally we show that a Bitonic[W] Counting Network is in fact quasi-linearizable such that its count operation quasi factor is bounded by $N * W$, where N is the number of working threads, and W is the width of the counting network.

2 Quasi Linearizable, definition

2.1 Lineariazablity Review

Definition 1. History: *Following [5] a history is a list of events which are ordered according to the time line in which they occurred, each event represents*

either a method invocation or a method response, a method invocation event is represented by the tuple $< O.method(args), T >$*, where* O *is the object the invocation operates on, method is the invoked method, args are the invocation arguments and* T *is the thread that started this invocation. Method invocation response is represented by the tuple* $< O : t(results), T >$*, where* t *is either OK or an exception name and results are the invocation result set. A response matches a prior invocation if it has the same object and thread, and no other events of* T *on object* O *appear between them.*

History H is called sequential if the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response.

An invocation is pending in H if no matching response follows the invocation. An extension of history H is a history constructed by appending zero or more responses matching the pending invocation of H. $Complete(H)$ is the sub-sequence of H consisting of all matching invocation and responses, thus, removing all pending invocations from H. $H|T$ is a history consisting of all and only the events of thread T in history H, two histories H and H' are *equivalent* if for each thread T, $H|T = H'|T$.

Definition 2. Linearizability *A history* H *is linearizable if it has an extension* H' *and there is a legal sequential history* S *such that:*

1. $Complete(H')$ *is equivalent to* S*.*
2. *If method invocation* m_0 *precedes method invocation* m_1 *in* H*, then the same is true in* S*.*

We first define *quasi-sequential specification* and then define what a *quasi-linearizable history* is and finally define a *quasi-linearizable* data structure. The definition follows the notations and standard model as in "The Art of multiprocessor programming" [4].

For example, consider the following sequential history of a queue: $H = enq(1)$, $enq(2)$, $deq()$=2, $enq(3)$, $deq()$=1, $deq()$=3. This sequential history is not legal for a queue, however, it is not "far" from being legal, by exchanging $enq()$=2 with $enq()$=1, one can get a legal sequential history. To formally define this reordering of H and to express how "far" is H from a legal sequential history, we introduce the quasi-linearizable concept.

A *sequential* history is an alternating sequence of invocations and responses, starting with an invocation, and each response matches the preceding invocation. We substitute these two matching events by a single event:
$< O.method(args), t(results) >$ (We ignore the thread executing this method call since it is redundant in a sequential history).

Unless specified otherwise all the *sequential* histories in the sequel are condensed in that way. Each event in such a history represents the tuple corresponding to both the invocation and the matching response.

A *sequential specification* is the set of all possible sequential runs of an object, each of these runs can be represented as a sequential history. The term *legal*

sequential history specifies that a sequential history is part of the sequential specification of the object that generated that history.

Definition 3. – *For each event e in a sequential history H, we define* $H[e]$ *to be its index in the history, clearly for two events, e and* e', $H[e'] < H[e]$ *iff* e' *is before e in H.*
- H_i *is the ith element in H. I.e.,* $H_{H[e]} = e$.
- $Events(H)$ *is the set of all the events in H.*
- $Distance(H', H)$ *the distance between two histories H and* H'*, such that* H' *is a permutation of H, is* $\max_{e \in Events(H)}\{|H'[e] - H[e]|\}$. .

Notice that by definition, the distance is defined only for histories which are a permutation of each other.

Definition 4. Object domain: *The set of all possible operations that are applicable to an object. We distinguish between operations that have different arguments or different returned values. For example, for* O *=stack,* $Domain(O) = \{< O.push(x),\ void >, < O.pop()\ x > | x \in X\} \cup \{< O.pop(), \phi >\}$*, where X is the set of all the possible elements in the stack.*

A sequential history $H|D$, is the projection of history H on a subset D of the events, i.e., H after removing from it all the events which are not in D. $H|O$ $= H|Domain(O)$.

We extend the sequential specification of an object O to a larger set that contains sequential histories which are not legal but are at a bounded "distance" from a legal sequential history. In other words, a sequential history H is in this set if there is some legal sequential history that its "distance" from H is bounded by some specified bound. We define that bound using a function that we name the *Quasi-linearization factor*. It is a function that operates on subsets of the object domain, mapping each subset to its "quasi factor", which is the upper bound on the relative movement among the operations in the subset that turn it into a legal sequential history. Formally,

Definition 5. Quasi-linearization factor: *A function* Q_O *of an object O defined as* $Q_O : D \rightarrow \mathbb{N}$[1]. *D is the set containing subsets of the object's domain, formally* $D = \{d_1, d_2, \ldots\} \subset Powerset(Domain(O))$[2]

Definition 6. Q_O**-Quasi-Sequential specification:** *is a set of all sequential histories that satisfy the "distance" bound implied by the quasi-linearization factor* Q_O *of an object O. Formally, for each sequential history H in the set, there is a legal sequential history S of O such that H is a prefix of* some *history* H' *which is a permutation of S and* $\forall$ *subset* $d_i \in D$: $Distance(H'|d_i, S|d_i) \leq Q_O(d_i)$

Definition 7. *Let* $Objects(H)$ *be the set of all the objects that H involves with.*

[1] The quasi-linearization factor range $\mathbb{N}$ is extended to a more robust set in the sequel.
[2] $d_1, d_2, \ldots$ are not necessarily disjoint sets, The quasi factor for operations that do appear in D is unbounded.

Definition 8. Q-Quasi-Linearizable history: *A history H is Q-Quasi-Linearizable if it has an extension H' and there is a sequential history S' such that:*

1. $Q = \bigcup_{O \in Objects(H)} Q_O$. [3]
2. *$Complete(H')$ is equivalent to S'.*
3. *If method invocation m_0 precedes method invocation m_1 in H, then the same is true in S'.*
4. *$\forall O \in Objects(H) : S'|O$ is member of the Q_O-Quasi-Sequential specification.*

We notice that a linearizable history H has Q-quasi-linearizable factor 0 for all of the domains of the objects that appear in it, i.e., for each object O in H, $Q(Domain(O)) = 0$.

Definition 9. Q-quasi-linearizable object: *An object implementation A is Quasi-Linearizable with Q if for every history H of A (not necessarily sequential), H is Q-Quasi-Linearizable history of that object.*

For example, consider the following quasi-linearization factor for a blocking queue implementation which is Q_{queue}-Quasi-Linearizable:
$D_{enq} = \{< O.enq(x), void > |x \in X\}$, $D_{deq} = \{< O.deq(), x > |x \in X\}$
Domain(Queue)=$D_{enq} \cup D_{deq}$

- $Q_{queue}(D_{enq}) = k$
- $Q_{queue}(D_{deq}) = 0$

Practically it means that an enqueue operation can bypass at most k preceding enqueue operations (and an arbitrary number of dequeue operations occurring in between). This quasi-linearizable queue specifications may be used as the task queue in the SEDA[11] system described in the Introduction.

H is a prefix of some history? Consider the following history for a concurrent counter:
$H =< getAndInc(), 3 >, < getAndInc(), 1 >$ This history can never be reordered to a legal sequential history since the event $< getAndInc(), 2 >$ is missing. However, it is reasonable for an execution of a quasi-linearizable implementation of a counter to create such a history because the execution can be stopped at any time. By appending the missing response and invocation $< getAndInc(), 2 >$ at the end of H we can reorder this history to a legal sequential history. This addition of unseen future events is described in the definition by adding a sequential history H to the quasi-sequential specification of the counter object if it is a prefix of *some* history which that history is equivalent to a legal sequential history, the *some* history is $H \cup < getAndInc(), 2 >$. If we do not allow completion of unseen events, hence do not place H in the quasi-sequential specification of the counter, we reduce the definition strength

[3] Q is a union of all the different object quasi-linearizable factors, each object has its own separate domain even for objects of the same type.

since any implementation would have been forced to return the entire range of getAndInc() results for not yet terminated operations (without skipping any numbers as the return value) in order to be quasi-linearizable, which in fact makes it similar to quiescent consistent, for instance, a single thread operating on the object has to get a fully linearizable contract from the implementation. It is important to notice that by adding non existing future events, it is not possible to make any history quasi-linearizable. For instance, if the quasi factor for the getAndInc() operation is 5, the following history $H =< getAndInc(), 8 >$ can never be transformed to a legal sequential history only by adding any future events, that is because no matter what unseen future events are added, the first event will need to be moved at least by distance 7 in a legal sequential history (because there are 7 events that must occur before it in any legal sequential history).

Distance measured on each subset of the domain separately The distance is measured only on the projection of the entire history on a subset of the domain, this is done intentionally since some operations may have no effect on others and we do not want to take them into account when we calculate the distance, For instance:
$H = enq(1)$, $size()$=1, $size()$=1,..., $size()$=1, $enq(2)$, $deq()$=2, $deq()$=1.
If we measure the distance on the enqueue operation and consider the $size()$=1 operations between $enq(1)$ and $enq(2)$, then the distance is unbounded, since an unbounded number of *size* operations may be executed (in this case one should consider a subset containing all possible *enq* operations separately). Another notion is that the subsets of the domain that has a quasi factor are not necessarily disjoint, which can be used to define a more generic quasi state. For instance it may be interesting to disallow reordering between *size* and *enqueue* operations, but to allow a reorder between *enqueue* and *dequeue* operations.

Extend Quasi Linearizable factor bound In the definition we have specified that the bound for each domain subset is a constant number, however, in some cases (as shown later on the Bitonic Counting Network), the bound can vary depending on different parameters such as configurable implementation parameters or different use cases (i.e., the number of threads accessing the object concurrently or different system properties). This is addressed by providing the ability to specify a custom function as the bound instead of a constant bound, and that function arguments take the parameters mentioned above. Formally, instead of having $Q_O : D \rightarrow \mathbb{N}$ we change the function as follows: $Q_O : D \rightarrow F^N$ where F^N is the set of all functions into $\mathbb{N}$ [4]. This way, a function f that represents a bound of a domain subset can receive the above variables as its parameters.

Predicting the future? Consider the following history for a concurrent queue:
$H = enq(1)$, $enq(2)$, $deq()$=3, $enq(3)$, $deq()$=1, $deq()$=2.

[4] F^N=$\{f$—f is a function and $Range(f) = \mathbb{N}\}$

By the definition of quasi-linearizability, this history is quasi-linearizable for $Q(D_{enq}) \leq 2$, however, it may seem weird that we consider this history legal because the first dequeue operation returns an element which has not yet been enqueued. However, practically speaking, if there was an implementation of a concurrent queue that this history represents an execution of it, it would mean that the implementation is predicting the future, which obviously is not feasible. The only type of such implementation would be one that returns a random value on its dequeue operation. However, for a data structure implementation to satisfy quasi-linearizability, *all* of its possible execution histories must be quasi-linearizable and given an implementation that returns a random result, we can easily schedule one execution example which may never be transformed to a legal sequential history while keeping the quasi distance boundaries.

Locality (Composition) Following [5], a property P of a concurrent system is said to be local if the system as a whole satisfies P whenever each individual object satisfies P. As shown in [5], linearizability is a local property, that is a history H is linearizable if and only if, $\forall O \in Objects(H) : H|O$ is linearizable.

Theorem 1. *H is Q-Quasi-Linearizable if and only if, $\forall O \in Objects(H) : H|O$ is Q_O-quasi linearizable.*

Sketch of Proof: It is easy to see that if the entire history H is quasi linearizable, then the projection of it on each object is quasi-linearizable by definition. In order to prove the other direction we need to show that given a history H, such that $\forall O \in Objects(H) : H|O$ is Q_O-quasi-linearizable, H is Q-quasi-linearizable. For each object O, we denote H'_O as the extension of $H|O$ implied by the quasi-linearizable definition of $H|O$ and S'_O as the sequential history that it is part of the Q_O-Quasi-sequential specification of O such that $Complete(H'_O)$ is equivalent to S'_O. By definition, S'_O is a prefix of some history which is a permutation of a legal sequential history, we denote that legal sequential history by S_O. We define $H' = H \bigcup_{O \in Objects(H)} H'_O$, clearly H' is an extension of H. We construct S by replacing all of the objects sub-histories $Complete(H'|O)$ with S'_O, clearly S is equivalent to $Complete(H')$ and the order of method invocations is kept between the two histories. We need to show that $\forall O \in Objects(H) : S|O$ is part of the Q_O-Quasi-Sequential specification of O, we get the above since $S|O = S'_O$ by construction. □

Composition is important in order to be able to use Quasi-Linearizable objects in a bigger system while keeping the Quasi-Linearizable property of the entire system. For instance, consider a system that keeps track of some internal components and operations and at some point needs to calculate the total number of operations executed on the system. Normally, such a system uses a linearizable shared counter that counts each of the operations occurrences, and a combined display counter that represents the total number of operations that is calculated by summing up all operation counters. Assume we have 2 counters for 2 different operations, we get the total number of operations by adding this two counters, assume this counters have a k_1, and k_2 respectively constant

quasi-linearizable bounds for their *add* method. From the composition derives that the quasi bound for the combined counter is $k_1 + k_2$ since the bound is kept for each counter upon composition. (If $k_1 = k_2 = 0$ we get a fully linearizable combined counter).

Operations that are invisible The definition treats all operation as equal, however, operations that are invisible (i.e, do not change the state of the object) can pose difficulties on actual implementations if they affect the distance equally as visible operation since the implementation will probably need to update some internal state for each of these operation in order to comply with the distance bounds. For instance, consider a queue that supports 3 operations: enqueue, dequeue and size, size in this case is considered as an invisible operation. There are a few natural ways to define such a quasi-linearizable queue, one would be to put the enqueue and size operations in the same domain subset in the quasi-linearization factor specification, as well as the dequeue and size operations, thus disabling the queue to return the wrong size value at any stage. However, this boundaries take size operations into consideration when calculating the distance of reordered enqueue operations. An alternative would be to put size in a separate domain subset, however, this will result in legal quasi-linearizable implementations that return size that was legal at some state. Intuitively, the distance between two visible operations should not be affected by invisible operation executed between the two. On the hand, there is still a need for a bound on the reordering distance of invisible operation, otherwise one cannot pose any limitations for this type of operations in a quasi-linearizable object. In order to address this, we can extend the *Distance* of two histories H and H' in the following way:

- Let $VEvent(H)$ be all the events that appear in H that are visible.
- Let $IEvent(H)$ be all the events that appear in H that are invisible.
- $Event(H) = VEvent(H) \cup IEvent(H)$
- $VDistance(H, H') = \max_{e \in VEvents(H)}\{|H'|VEvents(H)[e] - H|VEvents(H)[e]|\}$.
- $NDistance(H, H') = \max_{e \in IEvents(H)}\{|H'[e] - H[e]|\}$.
- $Distance(H, H') = \max\{NDistance(H, H'), VDistance(H, H')\}$.

Using this upgraded distance definition, the enqueue and size operations can be placed together in the same subset and also the dequeue and size operations, while we consider size to be an invisible operation.

Timed consistency comparison Timed consistency[9] adds the notion of time to the occurrences of events and not just order, roughly speaking, timed consistency models require that if a write operation is executed at time t, it must be visible to all processes by time $t + \Delta$. In that sense it has some similarity to the quasi-linearizable model, however, the concept of time is not equivalent to the concept of distance in the quasi-linearizable model. Specifically the timed consistency model does allow reordering of events. For example, consider the

quasi-linearizable queue mentioned before, in a scenario where each enqueue and dequeue operations are invoked with a time interval of Δ in between (no concurrent invocations), in order for an implementation to be timed consistent, it will have to return the serial order of events like a regular linearizable implementation. On the other hand, a quasi linearizable implementation does not, since the distance is not affected by time. In the sequel we show implementations which are quasi-linearizable but not timed consistent. Another difference is that timed consistency split operations into two groups, write or read. While quasi-linearizable separates operations according to their logical meaning.

3 Random Dequeued Queue

We offer a simple quasi linearizable non blocking queue implementation, illustrated in Figure 1, that behaves as follows: dequeue operation may return results not in the precise order they were enqueued (up to a constant bound) but when an empty (null) dequeue result is returned, there are no enqueued items in the queue. Formally, we describe this behavior with the following quasi-linearizable factor:

- $Q_{nb-queue}(D_{deq} \cup \{< deq(), null >\} = 0$ (no reordering of dequeue operations is allowed).
- $Q_{nb-queue}(D_{enq}) = k$ (we reorder enqueue operations up to distance k, which dictates the actual dequeue order).
- $\forall x \in X : Q(\{< enq(x), void >, < deq(), null >\}) = 0$ (enqueue operation can not be reordered over an empty dequeue operation)

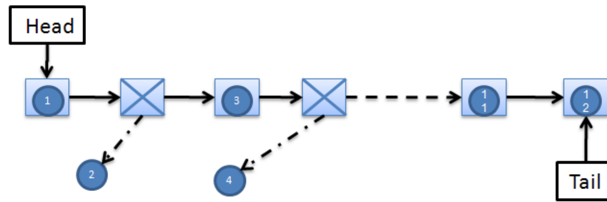

Fig. 1. Random Dequeue Queue

The idea is to spread the contention of the dequeue method by allowing to dequeue an element which is not at the head of the queue, but not more than k places away from the head. We base our quasi queue implementation on [8] which is based on a linked list, in fact our enqueue operation is exactly the same. We change the dequeue operation to pick a random index between 0 and k (the quasi factor), if the picked index is larger than 0 it iterates over the list from the head to the item at the specified index, it attempts to dequeue it by doing a single CAS(compare and set) which attempts to mark it as deleted. If failed it retries a few times and eventually falls back to the scenario as if index 0 is

picked. If it succeeds, this is the dequeued item. If the selected number is 0, the operation iterates over the list from the head until it finds a node which has not yet been dequeued. While iterating it attempts to remove all encountered dequeued nodes by attempting to advance the head of the list using a CAS each step. The implementation's code and proof of its quasi-lineariazbility property is omitted due to space limitations, for the code and a sketch of proof see [1] for online version of this paper.

4 Segmented Queue

The previous implementation of a quasi linearizable queue only reduces contention on the dequeue operation while the *enqueuer* threads still compete over the tail reference trying to enqueue new elements. Additionally, a dequeue operation iterates over its randomly selected number of nodes, while it may traverse over a node that it can dequeue along the way. In the following section we present an algorithm, illustrated in Figure 2, that scatters the contention both for dequeue and enqueue operations and in the normal case, iterates over less nodes while still keeping a constant quasi factor.

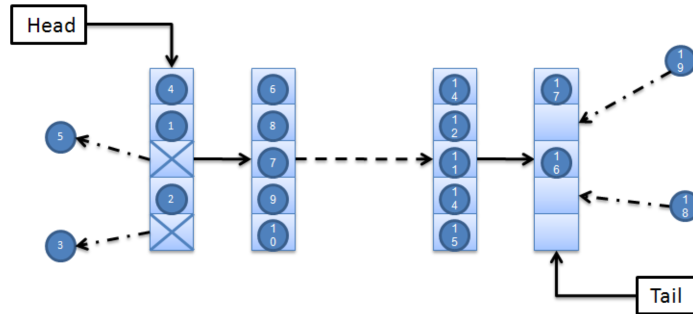

Fig. 2. Segmented Queue

The general idea is that the queue maintains a linked list of segments, each segment is an array of nodes in the size of the quasi factor (specified by *quasiFactor+1* in the implementation details), and each node has a deleted Boolean marker, which indicates if it has been dequeued. Each enqueuer iterates over the last segment in the linked list in some random permutation order; When it finds an empty cell it performs a CAS operation attempting to enqueue its new element. In case the entire segment has been scanned and no available cell is found (implying that the segment is full), then it attempts to add a new segment to the list.

The dequeue operation is similar, the dequeuer iterates over the first segment in the linked list in some random permutation order. When it finds an item which has not yet been dequeued, it performs a CAS on its deleted marker in order to "delete" it, if succeeded this item is considered dequeued. In case

the entire segment was scanned and all the nodes have already been dequeued (implying that the segment is empty), then it attempts to remove this segment from the linked list and repeats the process on the next segment. If there's no next segment, the queue is considered empty.

Based on the fact that most of the time threads do not add or remove segments, most of the work is done in parallel on different cells in the segments. This ensures a controlled contention depending on the segment size, which is the quasi factor.

```
public void enq(Object value){
  AtomicReference<Node>[] lastSegment = getLast();
  Node newNode = new Node(value);
  if (lastSegment == null){
    //Queue has no segments, create a new segment
    lastSegment = createLast(null);
  }
  while(true){
    int[] permutation = getRandomPermutation();
    for(int i = 0; i <= quasiFactor; ++i){
      final int index = permutation[i];
      //Cell is not empty, continue
      if (lastSegment[index].get() != null)
        continue;
      //Found empty cell, try to enqueue here
      if (lastSegment[index].compareAndSet(null, newNode))
        return;
    }

    //If reached here, no available position, create a new segment
    lastSegment = createLast(lastSegment);
  }
}
```

The method *createLast(lastSegment)* creates and adds a new last segment only if the current last segment is the provided method argument (lastSegment). The result of the method will be the current last segment which was either created by this invocation or another invocation if the current last segment is different than the method argument.

```
public Object deq(){
  AtomicReference<Node>[] firstSegment = getFirst();
  while(true){
    boolean hadNullValue = false;
    if (firstSegment == null)
      //Queue is empty
      return null;
```

```
    int[] permutation = getRandomPermutation();
    for(int i = 0; i <= quasiFactor; ++i){
      Node node = firstSegment[permutation[i]].get();
      //Check if this cell is empty, which means
      //an element can be enqueued to this cell in the future
      if (node == null) {
        hadNullValue = true;
        continue;
      }
      //Check if can dequeue node at index
      if (node.deleted.compareAndSet(false, true))
        return node.value;
    }
    //scanned the entire segment without finding a candidate
    //to dequeue

    //If there was an empty cell, the queue is considered empty
    if (hadNullValue)
      return null;
    //All nodes have been dequeued, we can safely remove the
    //first segment
    firstSegment = removeFirst(firstSegment);
  }
}
```

The method *removeFirst(firstSegment)* removes the first segment only if the current first segment is the provided method argument (firstSegment). The result of the method will be the current first segment.

The proof of the quasi-lineariazbility property of the implementation is omitted due to space limitations, for a sketch of proof see [1] for online version of this paper.

4.1 Segmented Stack

The above algorithm can be adapted in order to implement a stack. The difference is that when a push operation needs to add a new segment, it is added to the head of the list instead of the tail and a pop operation is similar to a dequeue operation.

5 Performance Evaluation

We evaluated the performance of our new algorithms on a Sun UltraSPARC T2 Plus multicore machine. This machine has 2 chips, each with 8 cores running at 1.2 GHz, each core with 8 hardware threads, so 64 way parallelism on a processor and 128 way parallelism across the machine. There is obviously a higher latency when going to memory across the machine (a two fold slowdown).

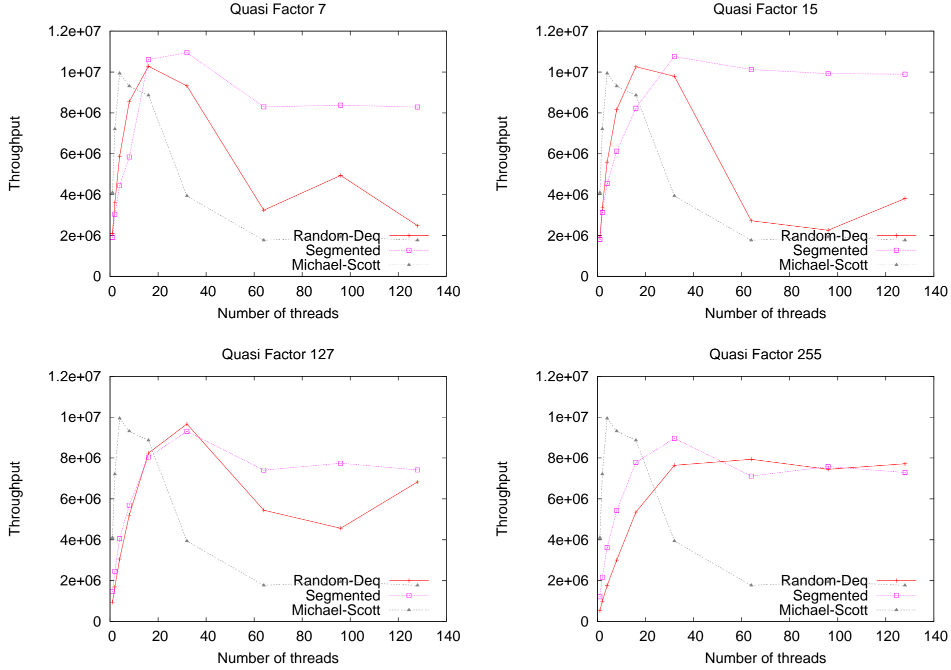

Fig. 3. Concurrent queue benchmark results

We can see from Figure 3 that the Segmented queue implementation out perform both Michael and Scott[8] and Random Dequeue when the number of threads increases, which is reasonable since it spreads both enqueue and dequeue contention. However as we increase the quasi factor, the overhead of scanning an entire segment just to realize the enqueuer needs to create a new segment or the dequeuer needs to remove the first segment, is increasing. On the other hand the Random Dequeue behaves very similar to the Michael and Scott algorithm when the quasi factor is low, but on high number of threads it improves if we increase the quasi factor, which is because the contention is reduced on the dequeue operation.

6 Bitonic[W] Counting Network

Next we show that the Bitonic[W] Counting Network (W the network width) is Q-quasi-linearizable, $Q(D_{inc} = \{< O.getAndInc(), n > | n \in \mathbb{N}\}) \leq N * W$ (where N is the number of working threads). For example for the Bitonic[4] Counting Network showed in Figure 4 with $N = 4$ we show that $Q(D_{inc}) \leq 16$.

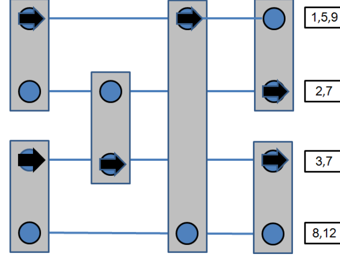

Fig. 4. Bitonic[4] Counting Network

For the Bitonic Counting Network we choose the quasi linearization points when a thread is increasing the counter at the exit from the network, we denote this history as S'.

Lemma 1. *For any n, in S' all the operations $\{< O.getAndInc(), m > | m \leq n - (N * W)\}$ precede $< O.getAndInc(), n >$.*

Proof: *Assume in contradiction that the lemma does not hold, denote by $< O.getAndInc(), m >$, $m < n - (N * W)$ as the missing operation. From the quiescent consistency property of Bitonic network, we know that if we will now schedule an execution that lets all the threads that are currently traversing the network (executing getAndInc operation) to finish their current operation, and prevent new threads or the same threads to reenter the network, the network must be at a legal state, that is, all the values up to n have been returned. From the algorithm of the network, we know that if $< O.getAndInc(), m >$ has not been executed yet, so does $\forall i > 0 :< O.getAndInc(), m+W*i >$ (because this operations are diverted to the same counter). From that we get that $< O.getAndInc(), m >$, $< O.getAndInc(), m + W >$,...,$< O.getAndInc(), m + W * (N - 1) >$ have not happened by the time that $< O.getAndInc(), n >$ occurred. Since there are at most $N - 1$ threads that are pending execution completion, they can never fill in the missing N operations to close the gap, in contradiction to the quiescent consistency property of the Bitonic network.*

From the lemma we know that each getAndInc() operation had bypassed at most $N*W$ other getAndInc() operation, therefore we can find a legal sequential history S which satisfies $Distance(S'|D_{inc}, S|D_{inc}) \leq N * W$. In this part we have shown an upper bound of $N * W$ for getAndInc operation, but this is not necessarily a tight bound.

7 Conclusions

In this paper we have shown a more relaxed concurrent model for linearizability and a few actual implementations which take advantage of the new model and are more concurrent than the equivalent linearizable implementation. We have demonstrated this with a queue that supports enqueue and dequeue operations, we can see how this definition can be adapted to a queue that also supports a peek operation. One way to define its quasi-linearizablity, is by specifying the quasi factor parameters as follows: $Q(D_{enq} \cup D_{peek}) = k$ and $\forall x : Q(\{< deq(), x >$

$, < peek(), x >\})=0$, meaning that a dequeue and peek operations may return an item at distance k from the head of the queue, but a peek can not return an item which has been already dequeued. Additionally we have shown that the already known Bitonic counting network implementation is quasi linearizable. This model can be applied to specify other quasi linearizable objects, such as, stack, heap etc., and thus allows a more concurrent implementation of these objects.

Acknowledgements. We would like to thank Adam Morrison, Nir Shavit and Maria Natanzon for very productive and helpful discussions. This paper was supported in part by grants from Sun Microsystems, Intel Corporation, as well as a grant 06/1344 from the Israeli Science Foundation and European Union grant FP7-ICT-2007-1 (project VELOX).

References

1. Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. http://sites.google.com/site/gkorland/research.
2. M. Ahamad, P. W. Hutto, G. Neiger, J. E. Burns, and P. Kohli. Causal memory: Definitions, implementation and programming. Technical Report GIT-CC-93/55, Georgia Institute of Technology, 1994.
3. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, New York, NY, USA, 1990. ACM.
4. M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.
5. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
6. M. H. J. Aspnes and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.
7. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.
8. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, New York, NY, USA, 1996. ACM.
9. F. J. Torres-Rojas, M. Ahamad, and M. Raynal. Timed consistency for shared distributed objects. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 163–172, New York, NY, USA, 1999. ACM.
10. W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.
11. M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.