TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

Quasi-Linearizability Relaxed Consistency For Improved Concurrency

Dissertation submitted in partial fulfillment of the requirements for the M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Eitan Yanovsky

The research work for this thesis has been carried out at Tel-Aviv University under the supervision of Prof. Yehuda Afek

August 2011

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.

Many linearizable and optimized concurrent algorithms are available for known algorithms and data structures, such as, Queue, Tree, Stack, Counter and Hash-table. However, sometimes these implementations are used in a more relaxed environment, provided as part of larger design pattern where a relaxed linearizability suffices rather than a strict one.

This thesis provides 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.

Acknowledgements

Parts of this thesis are joint work with Guy Korland. Who have provided the initial motivation for the need of this work, additionally he have helped with the different queues implementation that appears in Chapter 4

Contents

1 Introduction **1**

1.1 Background 1

1.2 Related work 4

1.2.1 Quiescent consistency 4

1.2.2 Eventual consistency 4

1.2.3 Timed consistency 5

1.3 Summary of results 5

2 Quasi Linearizable, definition **6**

2.1 Lineariazablity Review 6

2.2 Quasi-Linearizability 8

2.2.1 Definition details 12

H is a prefix of some history? 12

Distance measured on each subset of the domain separately 12

Extend Quasi Linearizable factor bound 13

Predicting the future? 13

Locality (Composition) 14

Operations that are invisible 15

3 Related work **17**

4 Implementations **18**

4.1 Random Dequeued Queue 18

4.1.1 Implementation 18
4.1.2 Proof of Quasi-Linearizability 19
4.2 Segmented Queue 24
4.2.1 Implementation 24
4.2.2 Proof of Quasi-Linearizability 27
4.2.3 Segmented Stack 31
4.2.4 Future Work - Double Segmented Queue 31

5 Performance Evaluation **33**

6 Bitonic[W] Counting Network **35**

7 Conclusions **37**

Chapter 1

Introduction

1.1 Background

Linearizability[4] 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 result 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[6], Quiescent consistency[5], Causal consistency[1], Release consistency[2], Eventual consistency[9] and Timed consistency[8]. But, the semantics of these relaxed conditions is less intuitive and the results are usually unexpected from a layman point of view. In this thesis 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.

While Linearizability is the most intuitive condition (providing a serial perception), trying to maintain such strict definition often results in high contention bottlenecks. Despite these limitations Linearizability is today the most popular consistency model for high level data structures such as Map, Queue and Stack. The definition of linearizability is equivalent to the following:

- All function calls have a linearization point at some instant between their invocation and response.

- All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition.

This definition is intuitive and composable, making it useful in the implementations of concurrent linearizable Data-Structures from other linearizable objects. But, when using this model to write concurrent highly scalable data structures the results are often less than satisfactory in terms of performance and scalability.

For example, SEDA[10], 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[7] 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.[5] 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 may be sacrificed for better scalability. More than that, even getting a stale data for a short while might not be a problem. A similar behavior commonly happens with a Stack, a linearizable LIFO implementation can ordinarily be replaced with an almost LIFO implementation for better scalability. Such implementation may return for a pop operation an element which is "close" to the top of the stack but not necessarily the element at the top. In some usages, this weaker constraint may be acceptable for the gain of better scalability, for example consider a special form of resource pool where it is preferred to reuse the most recently used resource due to some higher cost of initialization of a resource which was not used recently. Such resource pool can use a stack to hold its resources and always take the resource from the top, or if replaced with the stack implementation mentioned above, it may take a resource which was used recently but not necessarily the last used resource.

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 but very close to it. That is, there is a linearization order in which every task t which is in the queue, may only be dequeued if there is no other task t' which is still in the queue such that t' has been originally enqueued into the queue before t, and after t' was enqueued, k or more other tasks were enqueued prior to t. This non strict FIFO order generates the desired behavior in which we can execute any of the first k tasks in the queue but we are not allowed to bypass the task at the front of the queue more than k times. 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 work 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.

1.2 Related work

Many models were offered as weaker alternatives to Linearizability Three of them are *Quiescent consistency*[5], *Eventual consistency*[9] and *Timed consistency*[8].

1.2.1 Quiescent consistency

This model 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 not separated by quiescent state may not occur in program order. E.g., A dequeue operation that overlaps two enqueue operations, enqueue x and then enqueue y, may return the second enqueued object - y.
3. Operations whose occurrence is separated by a quiescent state should appear in the order of their occurrence and may not be reordered.

An object is in a quiescent state if currently there is no pending or executing operation on that object.

1.2.2 Eventual consistency

This model 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.

These two models, in most cases, allow 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 following model does provide a special form of upper bound on the "inconsistency".

1.2.3 Timed consistency

This model 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 $t + \Delta$. As it will be discussed later, it has some similarity to the quasi-linearizable model we provide here, however, the concept of time is not equivalent to the concept of distance which we present in the quasi-linearizable model. As a result the actual implementations and guarantees they provide that satisfy either Timed consistency or quasi-linearizable conditions are very different.

1.3 Summary of results

The contributions of this thesis are, first, in Chapter 2 we provide a formal definition of *Quasi Linearizability* and treat the abnormalities that may arise from the definition. Second, Chapter 4 we demonstrate two implementations of a FIFO queue that utilize this definition. Third, in Chapter 5 we empirically evaluate these implementations showing much better scalability relative to the state of the art implementations. Finally at Chapter 6 we show that this definition is implicitly used in already existing algorithms by showing that a *Bitonic[W] Counting Network*[5] is in fact quasi-linearizable.

Chapter 2

Quasi Linearizable, definition

In this chapter we provide a formal definition for Quasi-Linearizability condition for concurrent objects which is an extension to the Linearizability [4] condition. At first we start with a short review of the Linearizability condition definition, the definition follows the notations and standard model as in "The Art of multiprocessor programming" [3]. After the review, we extend the condition definition into Quasi-Linearizability accompanied by various examples.

2.1 Lineariazablity Review

Definition 1. History: *Following [4] 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.*

Definition 2. *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.*

For example, consider the following sequential history of a queue:

$H_1 =< Queue.enq(x), T_1 >, < Queue.OK(), T_1 >, < Queue.enq(y), T_2 >, < Queue.OK(), T_2 >, <$

$Queue.deq(), T_1 >, < Queue.OK(x), T_1 >$

And a non sequential (concurrent) history of a queue:

$H_2 =< Queue.enq(x), T_1 >, < Queue.enq(y), T_2 >, < Queue.OK(), T_1 >, < Queue.OK(), T_2 >, < Queue.deq(), T_1 >, < Queue.OK(x), T_1 >$

A sequential history is considered legal if it is part of the sequential specification of the object, i.e it is a valid, single threaded sequential run of the object.

Definition 3. *An invocation is pending in H if no matching response follows the invocation.*

For instance, in the history:

$H_3 =< Queue.enq(x), T_1 >, < Queue.enq(y), T_2 >, < Queue.OK(), T_1 >, < Queue.deq(), T_1 >, < Queue.OK(x), T_1 >$, the invocation $< Queue.enq(y), T_2 >$ is pending in H_3.

Definition 4. *An extension of history H is a history constructed by appending zero or more responses matching the pending invocation of H.*

In the above example, the following two histories are an extension of H_3: H_3 it self by appending zero responses or $H_{3'} = H_3, < Queue.OK(), T_2 >$, by appending the missing response for the pending $< Queue.enq(y), T_2 >$ invocation.

Definition 5. *$Complete(H)$ is the sub-sequence of H consisting of all matching invocation and responses, thus, removing all pending invocations from H.*

In our case, $Complete(H_3) =< Queue.enq(x), T_1 >, < Queue.OK(), T_1 >, < Queue.deq(), T_1 >, < Queue.OK(x), T_1 >$

Definition 6. *$H|T$ is a history consisting of all and only the events of thread T in history H,*

- $H_2|T_1 =< Queue.enq(x), T_1 >, < Queue.OK(), T_1 >, < Queue.deq(), T_1 >, < Queue.OK(x), T_1 >$
- $H_2|T_2 =< Queue.enq(y), T_2 >, < Queue.OK(), T_2 >$

Definition 7. *two histories H and H' are* equivalent *if for each thread T, $H|T = H'|T$.*

We say that a method call m_0 precedes a method call m_1 in history H if m_0 finished before m_1 started: that is, m_0's response event occurs before m_1's invocation event.

Definition 8. 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 call m_0 precedes method call m_1 in H, then the same is true in S.*

2.2 Quasi-Linearizability

We first define *quasi-sequential specification* and then define what a *quasi-linearizable history* is and finally define a *quasi-linearizable* data structure.

For example, consider the following sequential history of a queue:

$H =< Queue.enq(1), T_1 >, < Queue.OK(), T_1 >, < Queue.enq(2), T_2 >, < Queue.OK(), T_2 >$
$, < Queue.deq(), T_2 >, < Queue.OK(2), T_2 >, < Queue.enq(3), T_1 >, < Queue.OK(), T_1 >, <$
$Queue.deq(), T_1 >, < Queue.OK(1), T_1 >, < Queue.deq(), T_2 >, < Queue.OK(3), T_2 >$

This sequential history is not legal for a queue, however, it is not "far" from being legal, by exchanging the order of enqueue of element 1 and 2, 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:

$< O.method(args), t(results) >$ while removing the executing thread indication. We can ignore the thread executing this method call in a sequential history because in a sequential history operations do not overlap and eventually we consider a sequential history legal if the order of events there can represent a single threaded execution. For example the above history H is written as follows in its sequential representation:

$H =< Queue.enq(1), OK() >, < Queue.enq(2), OK(), < Queue.deq(), OK(2) >,$
$< Queue.enq(3), OK() >, < Queue.deq(), OK(1) >, < Queue.deq(), OK(3) >$

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 9.

- *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', $H[e'] < H[e]$ if and only if 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 10. Object domain: *The set of all possible operations that are applicable to an object. We distinguish between operations that have different arguments or different return 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 stack elements.*

Definition 11. *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.*

Definition 12. *A sequential history $H|O$, is the projection of history H on an object O which is defined as $H|O = H|Domain(O)$.*

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

- *$d_1, d_2, \ldots$ are not necessarily disjoint sets, The quasi factor for operations that do not appear in D is unbounded.*
- *The range of the function Q_O which is $\mathbb{N}$ is extended to a more robust set in the sequel, but this suffices in order to grasp the general concept.*

Definition 14. 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 $d_i \in D$: $Distance(H'|d_i, S|d_i) \leq Q_O(d_i)$*

For example consider the previously mentioned history:
$H =< Queue.enq(1), OK() >, < Queue.enq(2), OK(), < Queue.deq(), OK(2) >,$
$< Queue.enq(3), OK() >, < Queue.deq(), OK(1) >, < Queue.deq(), OK(3) >$
And the following Quasi-linearization factor for the queue object (Q) in the history H:

- $D_{enq} = \{< Queue.enq(x), OK() > | x \in X\}$
- $D_{deq} = \{< Queue.deq(), OK(x) > | x \in X\}$
- $Q_{Queue}(D_{enq}) = 2$
- $Q_{Queue}(D_{deq}) = 0$

There exists a legal sequential history
$S =< Queue.enq(2), OK() >, < Queue.enq(1), OK(), < Queue.deq(), OK(2) >,$
$< Queue.enq(3), OK() >, < Queue.deq(), OK(1) >, < Queue.deq(), OK(3) >$
such that for $Distance(H|D_{enq}, S|D_{enq}) \leq 2$ (in our case the distance is exactly 1),
and $Distance(H|D_{deq}, S|D_{deq}) = 0$. From the above we get that H is part of the Q_Q-Quasi-Sequential specification because in this case we say that H is a prefix of the same history H (it

self), and H is a permutation of S and it satisfies the distance bounds implied by the quasi factor Q_{Queue}.

Definition 15. *Let $Objects(H)$ be the set of all the objects that H involves with.*

Definition 16. 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. *$Q = \bigcup_{O \in Objects(H)} Q_O$. 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.*
2. *$Complete(H')$ is equivalent to S'.*
3. *If method call m_0 precedes method call 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 17. 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:
$D_{enq} = \{< Queue.enq(x), void > | x \in X\}$, $D_{deq} = \{< Queue.deq(), x > | x \in X\}$
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[10] system described in the Introduction.

2.2.1 Definition details

H is a prefix of some history?

Consider the following history for a concurrent counter:
$H =< Counter.getAndInc(), 3 >, < Counter.getAndInc(), 1 >$ This history can never be reordered to a legal sequential history since the event $< Counter.getAndInc(), 2 >$ 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 $< Counter.getAndInc(), 2 >$ 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 $H \cup < Counter.getAndInc(), 2 >$. 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 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 $H =< Counter.getAndInc(), 8 >$ 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 needs 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, consider the following history (written in a not formal form):

$H = enq(1)$, $size()$=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 $\mathbb{N}$. Formally, F^N=$\{f$—f is a function and $Range(f) = \mathbb{N}\}$. 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.

By the definition of quasi-linearizability, this history is quasi-linearizable for $\mathrm{Q}(D_{enq}) \leq 2$, 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 [4], 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 [4], 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 18. *H is Q-Quasi-Linearizable if and only if, $\forall O \in Objects(H) : H|O$ is Q_O-quasi linearizable.*

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 such implementations 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 results 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 other 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.

Chapter 3

Related work

In the introduction we have specified a few other weakened concurrent consistency models, *Quiescent consistency*[5], *Eventual consistency*[9] and *Timed consistency*[8]. The one that have the most similarity to this work is the Timed consistency model, At this chapter we conduct a short comparison between Quasi-Linearizability and Timed consistency models and point out the similarities and the differences. Timed consistency[8] 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 not 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 has 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.

Chapter 4

Implementations

4.1 Random Dequeued Queue

4.1.1 Implementation

We offer a simple quasi linearizable non blocking queue implementation, given in Algorithms 1 and 2, and illustrated in Figure 4.1.

The idea of the implementation 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 [7] 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 it succeeds, this is the dequeued item. If failed it reselects a random index and retries the entire process. The alrogithm will retry the process a few times up to a pre-configured max retries count, eventually if the process did not succeed to dequeue a random element after all the retries, it falls back to the scenario as if index 0 is picked. If the selected index 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.

This implementation retains an "almost" FIFO behavior by allowing the following weaker con-

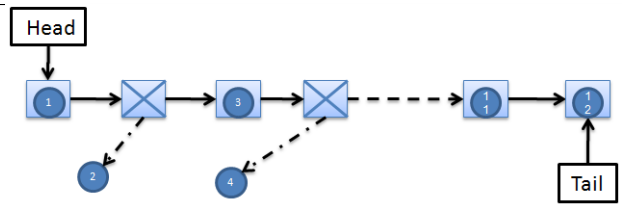

Figure 4.1: Random Dequeue Queue

straints: Dequeue operation may return results not in the precise order they were enqueued, but not too "far" from it (up to a constant bound). When an empty (null) dequeue result is returned, there are no remaining enqueued items in the queue. Formally, we describe this behavior with the following quasi-linearizable factor:

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

4.1.2 Proof of Quasi-Linearizability

In this part we show that the random dequeued queue implementation satisfies $Q_{nb-queue}$-Quasi Linearizability using the formal definition. Roughly speaking, we use a similar mechanism as in regular linearizability of selecting linearization point in the algorithm that determines the sequential full order between the overlapping and not overlapping method invocations. In this case we name these as quasi-linearization points. In full detail, we take a run of the algorithm and treat this run history as H, we need to show that the history H is $Q_{nb-queue}$-Quasi Linearizable. According to the definition we need to extend H to H' by adding responses for some pending method invocations. In order to do so we need to decide which pending methods invocation took place and which did not. We do that by selecting quasi-linearization points that consists of location and condition in the algorithm. We add a missing response in H' for a pending method only if its execution path have

```
input: x : value
// Allocate new node with the enqueued value
// Node structure {value:object, next:pointer, deleted:boolean}
newNode ← new Node();
newNode.value ← x;
while true do
    // tail :  A node pointing to the tail of the queue
    lastNode ← tail;
    nextNode ← lastNode.next;
    // Check that tail and lastNode are still consistent
    if lastNode = tail then
        // Was tail pointing to the last node?
        if nextNode =⊥ then
            // Try to link the node at the end of the list
            if CAS(lastNode.next, nextNode, newNode) then
                // Enqueue is complete, try to
                // swing tail to the inserted node
                CAS(tail, lastNode, newNode);
                break;
            end
        else
            // Tail was not pointing to the last node,
            // try to swing it to the next node
            CAS(tail, lastNode, nextNode);
        end
    end
end
```

Algorithm 1: Enqueue: Quasi linearizable queue

```
retries ← 0;
while true do
    // head :  A node pointing to the head of the queue
    first ← head, last ← tail , next ← first.next;
    // Check that first and head are still consistent
    if first = head then
        // Handle empty queue scenario
        if first = last then
            if next =⊥ then
                return ⊥;
            else
                // Tail is lagging behind, help it
                CAS(tail, last, next);
            end
        else
            randomIndex ← 0;
            if retries < MAX_RETRIES then
                // K : Predefined quasi factor
                randomIndex ← Random(K), retries ← retries + 1;
                if randomIndex > 0 then
                    value = AttemptRandomDequeue(randomIndex, next);
                    if value ≠⊥ then
                        return value;
                    end
                    // if failed repeat entire process
                end
                // iterate from head until encountered undeleted node
                while next ≠⊥ and next.deleted = true do
                    // Try to remove node by advancing head
                    if next ≠ last and CAS(head, first, next) then
                        first ← next, next ← next.next;
                    end
                    // Queue is empty
                    if next =⊥ then
                        return ⊥;
                    end
                    // Attempt to dequeue first element
                    if CAS(next.deleted, false, true) then
                        return next.value;
                    end
                end
            end
        end
    end
end
```

Algorithm 2: Dequeue: Quasi linearizable `queue`

```
   input: randomIndex : Integer, next : Node
1  i ← 0;
   // Move up to selected random index
2  while i < randomIndex do
3  |  if next.next =⊥ then
4  |  |  break;
5  |  end
6  |  next ← next.next, i ← i + 1;
7  end
   // Try to dequeue node at random index
8  if CAS(next.deleted, false, true) then
9  |  return next.value;
10 else
11 |  return ⊥;
12 end
```

Algorithm 3: AttemptRandomDequeue: Quasi linearizable `queue`

reached or passed a corresponding quasi-linearization point and met its condition. $Complete(H')$ consists only of non-pending methods of H and the pending methods that became non-pending by adding the missing responses in the process described in the previous. The quasi-linearization points of the algorithm are as follows:

Enqueue method (Algorithm 1):

- line 8 and the condition is a succesfully executed CAS which corresponds to adding the node at the end of the list.

Dequeue method (Algorithm 2):

- line 1 and the condition is that the method returns an empty queue state (line 5).
- line 9 and the condition is a successfully executed CAS which corresponds to dequeing the element that was marked as deleted by the CAS operation.
- line 21 and the condition is that next is NULL which corresponds to an empty queue result (line 25)
- line 28 and the condition is a successfully executed CAS which corresponds to dequeing the first element that was marked as deleted by the CAS operation.

After constructing $Complete(H')$ we need to show a sequential history S' that is equivalent to $Complete(H')$. We take each method invocation and its response in $Complete(H')$ and use the

corresponding linearization point to mark a single point in time that method affect takes place. Doing so we specify a full order between the methods and create a sequential history S'. S' is equivalent to $Complete(H')$ because it maintain the order of events per thread and for each method call m_0 that precedes method call m_1 in $Complete(H')$ the same is true in S' because the quasi-linearization points are inside each method invocation response time interval and the full order derived by them cannot change order of non overlapping methods. In our case we have one object in the history, a $nb-queue$, thus $S'|nb-queue = S'$. We need to show that S' is a member of the $Q_{nb-queue}$-Quasi-sequential specification

We recall that a history is in the $Q_{nb-queue}$-Quasi-sequential specification if it is a prefix of *some* sequential history which is a permutation of a legal sequential history of a non blocking queue, and the distance between that history and the legal sequential history satisfies the quasi-linearization factor. In our case we take S' it self to be the "*some* history" such that S' is a prefix of (every history is a prefix of it self), and show that S' is a permutation of some legal sequential history S such that $Distance(S'|D_{enq}, S|D_{enq}) \leq k$. From the algorithm, the enqueue order which is specified at S' according to the linearization points, also dictates the order of nodes that the linked list of elements is constructed of. If our implementation does not satisfy the distance for the enqueue operations, then for each legal sequential history S, $Distance(S'|D_{enq}, S|D_{enq}) > k$. This means that we can not reorder enqueue operation in any way in order to make it a legal sequential history, such that there is no enqueue operation that moved over k other enqueue operation during the reorder process of S' to a legal sequential history S. Since dequeue order is dictated by the enqueue order, the above is equivalent to that there is at least one dequeue operation in S' that returns an element which is placed inside the queue at a depth of at least $k+1$ (there are at least k elements before it including the head which are closer to the head of the queue). That means that the dequeue operation needs to traverse over at least k not deleted nodes until it reaches that element which it have dequeued. That contradicts the algorithm since if it picks 0 as a random index, in that case it returns the first not deleted node. Or if it picks an index between 1 and $k-1$, it iterates over that number of nodes from the head regardless of their deleted state, it could never reach a node at a distance of k from the head.

For simplicity we denote $D_{deq+} = D_{deq} \cup \{< deq(), null >\}$. In a non blocking queue sequential specification, the order of dequeue operation results is determined by the order of enqueue operation,

hence, it is enough to reorder just the enqueue operation in order to make it a legal sequential history. Therefore, we can transform S' to a legal sequential history S by just reordering the enqueue operation without reordering any dequeue operations and get the required distance for dequeue operations, hence $Distance(S'|D_{deq+}, S|D_{deq+}) = 0$.

We can see from the algorithm that an empty dequeue result is returned in both cases only when there was a point in time during the dequeue operation that there were no nodes in the list, that is the head was pointing to the tail and the next element was null. When this case happens, from the order of operations derived by the quasi-linearization points, prior to that event in S', there must be the same number of enqueue operations and dequeue operations that returned a non null result, otherwise there must be a node that is not deleted (dequeued). Therefore, there is no need to move any enqueue operations over a null dequeue operation in the reordering of S' to a legal sequential history. That satisfies the last bound, which is

$\forall x \in X :$

$Distance(S'|\{< enq(x), void >, < deq(), null >\}, S|\{< enq(x), void >, < deq(), null >\}) = 0.$

4.2 Segmented Queue

4.2.1 Implementation

The previous implementation of a quasi linearizable queue is quite simple and intuitive but it only reduces contention on the dequeue operation while the *enqueuer* threads still compete over the tail reference when 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. That implementation is quite simple and easy to learn from as an introduction example to the quasi-linearizable concept, moreover it fit to some scenarios, such as allocating tasks or ids from a previously filled pool of tasks and ids that were inserted to the queue in a not contentioned environment. The *dequeuers* in these scenarios are competing with each other to get tasks or ids. However, in a more general scenario, such as the SEDA [10], the *enqueuers* and *dequeuers* are working in parallel and the previous implementation still have high contention on the tail. In the following section we present an implementation given in Algorithms 4 and 6, and illustrated in Figure 4.2. This implementation 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.

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, if failed it repeats this process. 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 and repeat the previous process.

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, if failed it repeat this process. 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 is no next segment, the queue is considered empty.

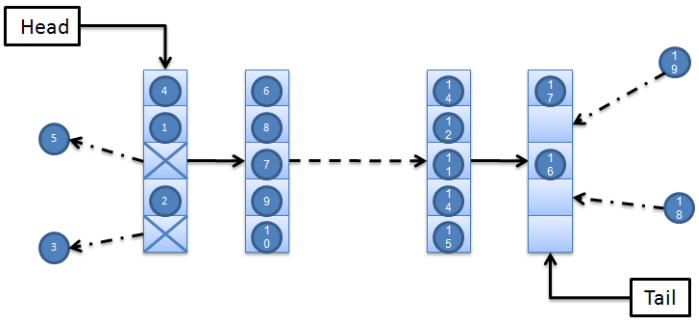

Figure 4.2: Segmented Queue

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.

The method *CreateLastSegment(expectedLastSegment)* (Algorithm 5) creates and adds a new last segment only if the current last segment is the provided method argument (expectedLast-Segment). The result of the method is the current last segment which was either created by this

Data types:
Node {value: data type, next: pointer, deleted: boolean}
Segment {nodes: array of size k of Node's, next: pointer, deleted: boolean }
Permutation {a permutation of the numbers 1 to k}

Shared variables:
tail : A segment pointing to the tail segment of the queue
head : A segment pointing to the head segment of the queue
K : Predefined quasi factor

Figure 4.3: Segmented queue definitions

```
input: x : value
// Allocate new node with the enqueued value
newNode ← new Node();
newNode.value ← x;
// Get last segment
lastSegment ← tail;
if lastSegment = ⊥ then
    lastSegment ← CreateLastSegment(⊥)
end
while true do
    // Iterate over the segment at a random permutation
    permutation ← GetRandomPermutation();
    for i ← 1 to K do
        randomIndex ← permutation.Get(i);
        if lastSegment.Get(randomIndex) = ⊥ then
            // Found empty cell, try to enqueue here
            if CAS(lastSegment.Get(randomIndex), ⊥, newNode) then
                return;
            end
        end
    end
    // If reached here, no empty position, create a new segment
    lastSegment ← CreateLastSegment(lastSegment);
end
```

Algorithm 4: Enqueue: Segmented Quasi linearizable `queue`

invocation or another concurrent invocation if the current last segment is different than the method argument.

```
   input: expectedLastSegment : Segment
 1 while true do
 2     lastSegment ← tail;
 3     nextSegment ← lastSegment.next;
       // Check if the expected last match,
       // otherwise a new segment was already added
 4     if lastSegment = expectedLastSegment then
           // Make sure lastSegment and tail still agree
 5         if lastSegment = tail then
               // Check if next segment after last segment do not exist
 6             if nextSegment = ⊥ then
                   // Prepare a new segment and attempt to
                   // insert it as the last segment
 7                 newSegment ← GetNewSegment();
 8                 if CAS(lastSegment.next, NULL, newSegment) then
 9                     return newSegment;
10                 end
11             end
12         end
13     end
       // Return the current last segment which
       // was added by another thread
14     return tail;
15 end
```

Algorithm 5: CreateLastSegment: Segmented Quasi linearizable `queue`

The method *GetNewSegment* that appears part of the *CreateLastSegment* method (Algorithm 5) can try to reuse segments from a pool that were previously generated using the *CreateLastSegment* but were not attached to the segments list due to failed CAS at line 65. In fact, we have done so in our java implementation that is used to benchmark this algorithm.

The method *RemoveFirstSegment(expectedFirstSegment)* removes the first segment only if the current first segment is the provided method argument (expectedFirstSegment). The result of the method is the current first segment.

4.2.2 Proof of Quasi-Linearizability

This queue implementation satisfies the previously defined $Q_{nb-queue}$-quasi linearizable properties. The general idea is that a dequeue operation is always done on the first segment until all the elements

```
firstSegment ← head;
while true do
    nullIndexes ← {};
    // Queue is empty
    if firstSegment = ⊥ then
        return ⊥
    end
    // Iterate over the segment at a random permutation
    permutation ← GetRandomPermutation();
    for i ← 1 to k do
        randomIndex ← permutation.Get(i);
        node ← firstSegment.Get(randomIndex);
        // Found a candidate for dequeue, try to delete it
        if (node ≠⊥) then
            if CAS(node.deleted, false, true) then
                return node.value;
            end
        else
            // Keep track of null nodes in order get
            // a snapshot of null nodes if needed
            nullIndexes.Add(randomIndex);
        end
        // Scanned the entire segment without finding
        // a candidate to dequeue If there were empty
        // cells, check if their state has changed
        if nullIndexes ≠ {} then
            foreach i in nullIndexes do
                // An element was enqueued to one of the previously
                // empty nodes, retry dequeue cycle
                if firstSegment.Get(i) ≠⊥ then
                    break;
                end
            end
            // Null indexes didn't change, we have
            // a legit snapshot, queue is empty
            return ⊥
        else
            // All nodes have been dequeued, we can safely remove
            // the first segment and retry dequeue
            firstSegment ← RemoveFirstSegment(firstSegment);
        end
    end
end
```

Algorithm 6: Dequeue: Segmented Quasi linearizable `queue`

```
while true do
    firstSegment ← head;
    tailSegment ← tail;
    next ← firstSegment.next;
    // Check for consistent variable state
    if firstSegment = head then
        // Check for empty queue state
        if firstSegment = tailSegment then
            // Check if next segment after first segment do not exist
            if nextSegment =⊥ then
                return ⊥;
            else
                // Tail is lagging behind, help it
                CAS(tail, tailSegment, next);
            end
        end
        result ← next;
        // If the first segment is the expected first
        // segment, try to remove it
        if value = expectedFirstSegment) then
            CAS(head, firstSegment,next);
            // We can successfully break loop because the expected
            // first segment was removed by this thread or another
            break;
        end
    end
end
// Return the current first element
return head;
```

Algorithm 7: RemoveFirstSegment: Segmented Quasi linearizable `queue`

of that segment have been dequeued, the same goes for the enqueue operation, each segment is fully filled before a new segment is created. The quasi-linearization points of the algorithm are as follows:

Enqueue method (Algorithm 4):

- line 15 and the condition is a successfully executed CAS which corresponds to inserting the node into the segment and enqueing the value.

Dequeue method (Algorithm 6):

- line 6 and the condition is the firstSegment is NULL which corresponds to an empty queue result (line 3).
- line 14 and the condition is the entire segment was scanned and there was at least one NULL node and all the non-NULL nodes were marked as deleted. This in turns correspond to an empty queue result (line 21) if the double scan of NULL nodes shows no difference in the NULL nodes state.
- line 11 and the condition is a successfully executed CAS which corresponds to marking a node as deleted and dequeue its value.

One can see that in order to create a legal sequential history from the sequential history derived by the quasi-linearization points after extending and completing it in a similar way of the previous implementation proof, the order of the enqueue operation in the derived sequential history would need to be reordered only with the corresponding dequeue events that belong to the same segment only, without crossing into other segments dequeue operation.

This assumption relies on the fact that according to the algorithm and quasi-linearization selected points, the corresponding enqueue events have all occurred either before other segments enqueue events or after other segments enqueue events without interleaving with other segments enqueue events. Since each segment length is of the quasi factor (k), this means that during the reordering of the derived sequential history in order to make it legal sequential history, the enqueue events in that history that belongs to the same segment, moves at most k places, as required.

There are two possible algorithm paths of the dequeue operation that returns an empty queue result. The first one at line 3, with the corresponding quasi-linearization point at line 6, may only

occur when there is no pending elements to dequeue otherwise the first segment could have not been null since the linearization point of the enqueue operation is only after there is at least one segment in the segments list. The second one at line 21, which we linearize at line 14, indicates that there was no available element to dequeue at that point because after iterating over the entire segment there was no undeleted node and only nodes without enqueues element. The double scan of the NULL nodes guarantee a snapshot of the NULL nodes states meaning that no new values were enqueued to the segment between the first iteration and the second.

4.2.3 Segmented Stack

The above algorithm can be adapted very easily in order to implement a non-blocking stack instead of a non-blocking queue. The difference is that when a push operation needs to add a new segment, it adds it to the head of the list of segments instead of the tail, thus having *CreateFirstSegment* instead of the *CreateLastSegment* method. And the pop operations is similar to the dequeue operation. Obviously this yields higher contention when adding and removing segments because both are competing on the same node in the segment list but most operations are still spread across a segment.

4.2.4 Future Work - Double Segmented Queue

During the evaluation of the performance of the current algorithm, shown in Chapter 5, we noticed that the performance of the segmented queue starts to decrease from some segment size (quasi factor). which is quite reasonable because the times it takes to scan a segment when it is already full or all its nodes are deleted (dequeued) increases because the segment size increases. Here we suggest an idea for future work of an algorithm that tries to improve that. The general idea is to extend the suggested algorithm to work on two segments each time instead of one, i.e a dequeue operation can randomly dequeue any element from the first two segments. Once all the items in the first segment are dequeued, it is removed in a similar way to the current algorithm and the dequeue operation chooses again an item at random from the first two segments. The enqueue operation can be adapted in a similar way, working with the last two segments. One point to consider is that instead of iterating at a random permutation each time, the algorithm should have a fall back mechanism that after a few failure attempts of randomly selecting a node from the first (or last)

two segments, it falls back to a deterministic iteration that starts from the first segment until an available node is found to dequeue (or enqueue), this allows the algorithm to detect that all the nodes in the first segment are already deleted (or full) and it can remove that segment (or add a new segment) safely. Obviously this kind of algorithms are more complicated and may evantually introduce more overhead due to the inherited complexity they hold.

Chapter 5

Performance Evaluation

We evaluated the performance of our new algorithms implemented in Java on a Sun UltraSPARC T2 Plus multi-core 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). The benchmark consist of even number of enqueuers and dequeuers threads, each thread executing the same number of operations thus having 50% enqueue and 50% dequeue operations.

We can see from Figure 5.1 that the Segmented queue implementation out perform both Michael and Scott[7] 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.

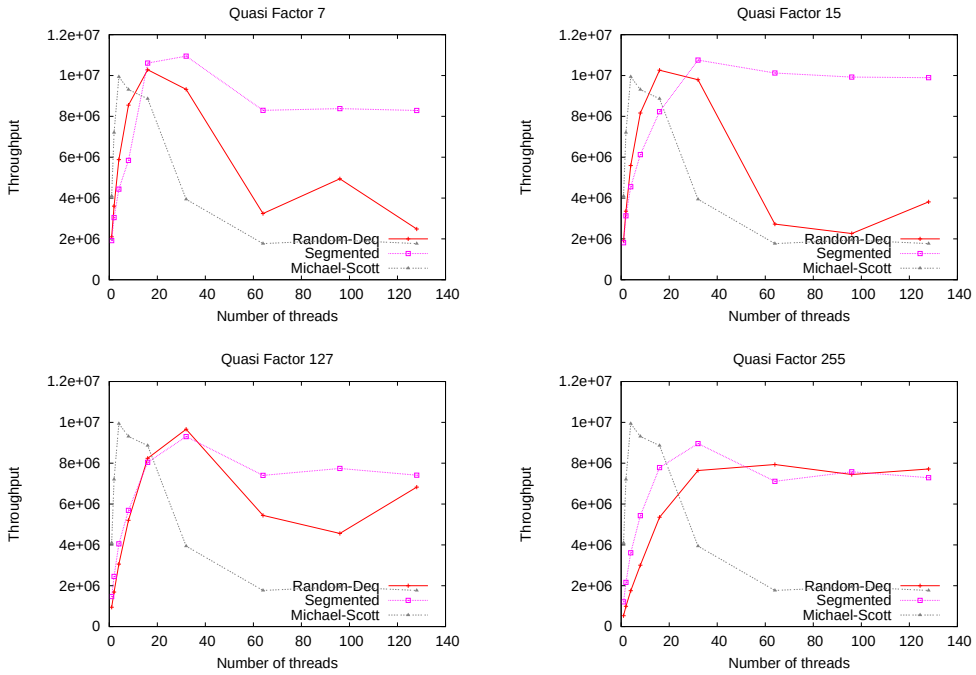

Figure 5.1: Concurrent queue benchmark results

Chapter 6

Bitonic[W] Counting Network

So far we have demonstrated the quasi-linearizable definition for queues and shown two algorithms of non blocking queues that satisfies quasi-linearizability. In this part we show the definition is quite generic and it fits to other data structures as well, in this case a shared counter. Moreover, we show that the already existing *Bitonic Counting Network*[5] which is an implementation of a counter, satisfies quasi-linearizability. Formally, A Bitonic[W] Counting Network (W the network width) is $Q_{counter} - quasi - linearizable$, $Q_{counter}(D_{inc} = \{< getAndInc(), n > | n \in \mathbb{N}\}) \leq N * W$ (where N is the number of working threads). For example for the Bitonic[4] Counting Network showed in Figure 6.1 with $N = 4$ we show that $Q(D_{inc}) \leq 16$.

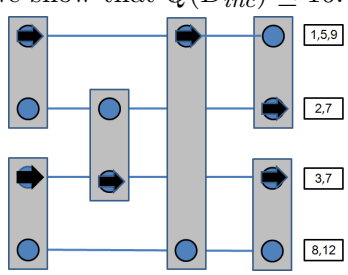

Figure 6.1: Bitonic[4] Counting Network

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 sequential history derived by the quasi linearization points as S' (The process of taking a concurrent history and making a sequential history out of it using the quasi-linearization point is desrcibed in details in Chapter 4 at section 4.1.2).

Lemma 19. *For any n, in S' all the operations $\{< getAndInc(), m > | m \leq n - (N * W)\}$ precede*

$< getAndInc(), n >$.

Proof: *Assume in contradiction that the lemma does not hold, denote by* $< getAndInc(), m >$, $m < n - (N * W)$ *as the missing operation. From the quiescent consistency property of Bitonic network, we know that if we 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* $< getAndInc(), m >$ *has not been executed yet, so does* $\forall i > 0 :< getAndInc(), m+W*i >$ *(because this operations are diverted to the same counter). From that we get that* $< getAndInc(), m >$, $< getAndInc(), m + W >$,...,$< getAndInc(), m + W * (N - 1) >$ *have not happened by the time that* $< getAndInc(), n >$ *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.

Chapter 7

Conclusions

In this work 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-linearizability, 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.

Bibliography

[1] 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.

[2] 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.

[3] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[4] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[5] M. H. J. Aspnes and N. Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[6] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[7] 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.

[8] 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.

[9] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[10] 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.

Appendix