Between Linearizability and Quiescent Consistency[⋆]
Quantitative Quiescent Consistency

Radha Jagadeesan and James Riely

DePaul University

Abstract Linearizability is the de facto correctness criterion for concurrent data structures. Unfortunately, linearizability imposes a performance penalty which scales linearly in the number of contending threads. Quiescent consistency is an alternative criterion which guarantees that a concurrent data structure behaves correctly when accessed sequentially. Yet quiescent consistency says very little about executions that have any contention.

We define quantitative quiescent consistency (QQC), a relaxation of linearizability where the degree of relaxation is proportional to the degree of contention. When quiescent, no relaxation is allowed, and therefore QQC refines quiescent consistency, unlike other proposed relaxations of linearizability. We show that high performance counters and stacks designed to satisfy quiescent consistency continue to satisfy QQC. The precise assumptions under which QQC holds provides fresh insight on these structures. To demonstrate the robustness of QQC, we provide three natural characterizations and prove compositionality.

1 Introduction

This paper defines *Quantitative Quiescent Consistency (QQC)* as a criterion that lies between linearizability [10] and quiescent consistency [3], [11], [17]. The following example should give some intuition about these criteria.

Example 1.1. Consider a counter object with a single `getAndIncrement` method. The counter's sequential behavior can be defined as a set of strings such as $[^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ (^{+}\ )^{+}_{2}$ where $[^{+}$ denotes an invocation (or call) of the method and $]^{+}_{i}$ denotes the response (or return) with value i. Suppose each invocation is initiated by a different thread.

A concurrent execution may have overlapping method invocations. For example, in $(^{+}\ [^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ )^{+}_{2}$ the execution of $(^{+}\ )^{+}_{2}$ overlaps with both $[^{+}\ ]^{+}_{0}$ and $\{^{+}\ \}^{+}_{1}$, whereas $[^{+}\ ]^{+}_{0}$ finishes executing before $\{^{+}\ \}^{+}_{1}$ begins. Consider the following four executions.

$$(^{+}\ [^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ )^{+}_{2} \qquad (^{+}\ \{^{+}\ \}^{+}_{1}\ [^{+}\ ]^{+}_{0}\ )^{+}_{2} \qquad [^{+}\ (^{+}\ )^{+}_{2}\ \{^{+}\ \}^{+}_{1}\ ]^{+}_{0} \qquad [^{+}\ (^{+}\ )^{+}_{2}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}$$

Linearizability states roughly that *every* response-to-invocation order in a concurrent execution must be consistent with the sequential specification. Thus, the first execution is linearizable, since the response of $[^{+}\ ]^{+}_{0}$ precedes the invocation of $\{^{+}\ \}^{+}_{1}$ in the specification. However, none of the other executions is linearizable. For example, the response of $\{^{+}\ \}^{+}_{1}$ precedes the invocation of $[^{+}\ ]^{+}_{0}$ in the second execution.

⋆ Research supported by NSF 0916741.
The full version of this paper is available at `http://arxiv.org/abs/1402.4043`.

Linearizability can also be understood in terms the *linearization point* of a method execution, which must occur between the invocation and response. From this perspective, the first execution above is linearizable because we can find a sequence of linearization points that agrees with the specification; this requires only that the linearization point of $(^{+}\ )^{+}_{2}$ follow that of $\{^{+}\ \}^{+}_{1}$. No such sequence of linearization points exists for the two other executions.

Quiescent consistency is similar to linearizability, except that the response-to-invocation order must be respected only across a quiescent point, that is, a point with no open method calls. The first three executions above are quiescently consistent because there are no non-trivial quiescent points. The last execution fails to be quiescently consistent since the order from $(^{+}\ )^{+}_{2}$ to $\{^{+}\ \}^{+}_{1}$ is not preserved in the specification.

We define *Quantitative Quiescent Consistency (QQC)* to require that the number of response-to-invocation pairs that are out-of-order at any point be bounded by the number of open calls that might be ordered later in the specification. We also give a *counting characterization* of QQC, which requires that if a response matches the i^{th} method call in the specification, then it must be preceded by at least i invocations.

The first two executions above are QQC; however, the last two are not. In the second execution, the open call to $(^{+}\ )^{+}_{2}$ justifies the return of $\{^{+}\ \}^{+}_{1}$ before $[^{+}\ ]^{+}_{0}$ since $(^{+}\ )^{+}_{2}$ occurs after $\{^{+}\ \}^{+}_{1}$ in the specification. However, in the third execution, the return of $(^{+}\ )^{+}_{2}$ before $\{^{+}\ \}^{+}_{1}$ cannot be justified only by the call to $[^{+}\ ]^{+}_{0}$ since $[^{+}\ ]^{+}_{0}$ occurs earlier in the specification. Following the counting characterization sketched above, the third execution fails since $(^{+}\ )^{+}_{2}$ is the third method call in the specification trace, but the response of $(^{+}\ )^{+}_{2}$ is only preceded by two invocations: $[^{+}$ and $(^{+}$. □

Quiescent consistency is too coarse to be of much use in reasoning about concurrent executions. For example, a sequence of interlocking calls never reaches a quiescent point; therefore it is trivially quiescently consistent. This includes obviously correct executions, such as $[^{+}\ (^{+}\ ]^{+}_{0}\ [^{+}\ )^{+}_{1}\ (^{+}\ ]^{+}_{2}\ [^{+}\ )^{+}_{3}\ (^{+}\ ]^{+}_{4}\ [^{+}\ \cdots$, nearly correct executions, such as $[^{+}\ (^{+}\ ]^{+}_{1}\ [^{+}\ )^{+}_{0}\ (^{+}\ ]^{+}_{3}\ [^{+}\ )^{+}_{2}\ (^{+}\ ]^{+}_{5}\ [^{+}\ \cdots$, and also ridiculous executions, such as $[^{+}\ (^{+}\ ]^{+}_{1074}\ [^{+}\ )^{+}_{17}\ (^{+}\ ]^{+}_{2344}\ [^{+}\ )^{+}_{3}\ (^{+}\ \cdots$.

Linearizability has proven quite useful in reasoning about concurrent executions; however, it fundamentally constrains efficiency in a multicore setting: Dwork, Herlihy, and Waarts [6] show that if many threads concurrently access a linearizable counter, there must be either a location with high contention or an execution path that accesses many shared variables. Shavit [14] argues that the performance penalty of linearizable data structures is increasingly unacceptable in the multicore age. This observation has lead to a recent renewal of interest in nonlinearizable data structures. As a simple example, consider the following counter implementation: a simplified version of the counting networks of Aspnes, Herlihy, and Shavit[3].

```
class Counter<N:Int> {
   field b:[0..N-1] = 0;                        // 1 balancer
   field c:Int[]    = [0, 1, ..., N-1]; // N counters
   method getAndIncrement():Int {
      val i:[0..N-1];
      atomic { i = b; b++; }
      atomic { val v = c[i]; c[i] += N; return v; } } }
```

The N-`Counter` has two fields: a *balancer* `b` and an array `c` of N integer counters. There are two atomic actions in the code: The first reads and updates the balancer, setting the local index variable `i`. The second reads and updates the `i`th counter. Although the balancer has high contention in our simplified implementation, the counters do not; balancers that avoid high contention are described in [3].

Example 1.2. The N-`Counter` behaves like a sequential counter if calls to `getAndIncrement` are sequentialized. To see this, consider a 2-`Counter`, with initial state $\langle \mathtt{b} = 0,\, \mathtt{c} = [0, 1]\rangle$. In a series of sequential calls, the state progresses as follows, where we show the execution of the first atomic with the invocation and the second atomic with the response. The execution $\texttt{[}^{+}\ \texttt{]}^{+}_{0}\ \texttt{\{}^{+}\ \texttt{\}}^{+}_{1}\ \texttt{(}^{+}\ \texttt{)}^{+}_{2}$ can be elaborated as follows.

$$\begin{aligned}
\langle \mathtt{b} = 0, \mathtt{c} = [0, 1]\rangle &\xrightarrow{\texttt{[}^{+}} \langle \mathtt{b} = 1, \mathtt{c} = [0, 1]\rangle \xrightarrow{\texttt{]}^{+}_{0}} \langle \mathtt{b} = 1, \mathtt{c} = [2, 1]\rangle \\
&\xrightarrow{\texttt{\{}^{+}} \langle \mathtt{b} = 0, \mathtt{c} = [2, 1]\rangle \xrightarrow{\texttt{\}}^{+}_{1}} \langle \mathtt{b} = 0, \mathtt{c} = [2, 3]\rangle \\
&\xrightarrow{\texttt{(}^{+}} \langle \mathtt{b} = 1, \mathtt{c} = [2, 3]\rangle \xrightarrow{\texttt{)}^{+}_{2}} \langle \mathtt{b} = 1, \mathtt{c} = [4, 3]\rangle
\end{aligned}$$

When there is concurrent access, the 2-`Counter` allows nonlinearizable executions, such as $\texttt{(}^{+}\ \texttt{\{}^{+}\ \texttt{\}}^{+}_{1}\ \texttt{[}^{+}\ \texttt{]}^{+}_{0}\ \texttt{)}^{+}_{2}$.

$$\begin{aligned}
\langle \mathtt{b} = 0, \mathtt{c} = [0, 1]\rangle &\xrightarrow{\texttt{(}^{+}} \langle \mathtt{b} = 1, \mathtt{c} = [0, 1]\rangle \\
&\xrightarrow{\texttt{\{}^{+}} \langle \mathtt{b} = 0, \mathtt{c} = [0, 1]\rangle \xrightarrow{\texttt{\}}^{+}_{1}} \langle \mathtt{b} = 0, \mathtt{c} = [0, 3]\rangle \\
&\xrightarrow{\texttt{[}^{+}} \langle \mathtt{b} = 1, \mathtt{c} = [0, 3]\rangle \xrightarrow{\texttt{]}^{+}_{0}} \langle \mathtt{b} = 1, \mathtt{c} = [2, 3]\rangle \\
&\qquad\qquad\qquad\qquad\qquad\quad \xrightarrow{\texttt{)}^{+}_{2}} \langle \mathtt{b} = 1, \mathtt{c} = [4, 3]\rangle
\end{aligned}$$

With a sequence of interlocking calls, it is also possible for the N-`Counter` to execute as $\texttt{[}^{+}\ \texttt{(}^{+}\ \texttt{]}^{+}_{1}\ \texttt{[}^{+}\ \texttt{)}^{+}_{0}\ \texttt{(}^{+}\ \texttt{]}^{+}_{3}\ \texttt{[}^{+}\ \texttt{)}^{+}_{2}\ \texttt{(}^{+}\ \texttt{]}^{+}_{5}\ \texttt{[}^{+}\ \cdots$, producing an infinite sequence of values that are just slightly out of order. Using the results of this paper, one can conclude that with a maximum of two open calls, the value returned by `getAndIncrement` will be “off” by no more than 2, but this does not follow from quiescent consistency. □

Our results are related to those of [2], [3], [5], [16]. In particular, Aspnes, Herlihy, and Shavit[3] prove that in any *quiescent* state (with no call that has not returned), such a counter has a “step-property”, indicating the shape of `c`. Between $\texttt{\}}^{+}_{1}$ and $\texttt{]}^{+}_{0}$ in the second displayed execution of Example 1.2, the states with $\mathtt{c} = [0, 3]$ do *not* have the step property, since the two adjacent counters differ by more than 1.

Aspnes, Herlihy, and Shavit[3] imply that the step property is related to quiescent consistency, but they do not provide a formal definition. It appears that they have in mind is something like the following: An execution is *weakly quiescent consistent* if any uninterrupted subsequence of *sequential* calls (single calls separated by quiescent points) is a subtrace of a specification trace.

The situation is delicate: Although the increment-only counters of [3] are quiescently consistent in the sense we defined in Example 1.1 (indeed, they are QQC), the increment-decrement counters of [2], [5], [16] are only *weakly* quiescent consistent. Indeed, the theorems proven in [16] state only that, at a quiescent point, a variant of the step property holds. They state nothing about the actual values read from the individual counters. Instead, our definition requires that a quiescently consistent execution be a permutation of *some* specification trace, even if it has no nontrivial quiescent points.

Example 1.3. Consider an extension of the 2-`Counter` with `decrementAndGet`.

```
method decrementAndGet():Int {
   val i:[0..N-1];
   atomic { i = b-1; b--; }
   atomic { c[i] -= N; return c[i]; } }
```

The execution $[^+\ \{^+\ (^-\ <^-\ >^-_{-2}\ ]^+_{-2}\ \}^+_1\ )^-_1$ is possible, although this is not a permutation of any specification trace. The execution proceeds as follows.

$$\begin{aligned}
\langle \mathsf{b}=0, \mathsf{c}=[0,1]\rangle &\xrightarrow{[^+} \langle \mathsf{b}=1, \mathsf{c}=[0,1]\ \rangle \xrightarrow{\{^+} \langle \mathsf{b}=0, \mathsf{c}=[0,1]\rangle \\
&\xrightarrow{(^-} \langle \mathsf{b}=1, \mathsf{c}=[0,1]\ \rangle \xrightarrow{<^-} \langle \mathsf{b}=0, \mathsf{c}=[0,1]\rangle \\
&\xrightarrow{>^-_{-2}} \langle \mathsf{b}=0, \mathsf{c}=[-2,1]\rangle \xrightarrow{]^+_{-2}} \langle \mathsf{b}=0, \mathsf{c}=[0,1]\rangle \\
&\xrightarrow{\}^+_1} \langle \mathsf{b}=0, \mathsf{c}=[0,3]\ \rangle \xrightarrow{)^-_1} \langle \mathsf{b}=0, \mathsf{c}=[0,1]\rangle
\end{aligned}$$
□

It is important to emphasize that this increment-decrement counter is not even quiescently consistent according to our definition. There is no hope that it could satisfy any stronger criterion.

Of course counters are not the only data structures of interest. In the full paper, we treat concurrent stacks in detail. We define a simplified N-`Stack` below; the full, tree-based data structure is defined in Shavit and Touitou[16].

```
class Stack<N:Int> {
   field b:[0..N-1] = 0;                      // 1 balancer
   field s:Stack[]  = [[], [], ..., []]; // N stacks of values
   method push(x:Object):Unit {
      val i:[0..N-1];
      atomic { i = b; b++; }
      atomic { val v = s[i].push(x); return v; } }
   method pop():Object {
      val i:[0..N-1];
      atomic { i = b-1; b--; }
      atomic { val v = s[i].pop(); return v; } } }
```

The trace given in Example 1.3 for the increment-decrement counter is also a trace of the stack, where we interpret `+` as `push` and `-` as `pop`. Whereas this is a nonsense execution for a counter, it is a linearizable execution of a stack: simply choose the linearization points so that each push occurs immediately before the corresponding pop. Nonetheless, the N-`Stack` is only *weakly* quiescent consistent in general.

Example 1.4. The N-`Stack` generates the execution $[^+_a\ ]^+\ (^+_b\ )^+\ \{^+_c\ <^-\ >^-_a\ \}^+$ as follows.

$$\begin{aligned}
\langle \mathsf{b}=0, \mathsf{s}=[[\,],[\,]]\ \rangle &\xrightarrow{[^+_a} \langle \mathsf{b}=1, \mathsf{s}=[[\,],[\,]]\ \rangle \xrightarrow{]^+} \langle \mathsf{b}=1, \mathsf{s}=[[a],[\,]]\ \rangle \\
&\xrightarrow{(^+_b} \langle \mathsf{b}=0, \mathsf{s}=[[a],[\,]]\ \rangle \xrightarrow{)^+} \langle \mathsf{b}=0, \mathsf{s}=[[a],[b]]\rangle \\
&\xrightarrow{\{^+_c} \langle \mathsf{b}=1, \mathsf{s}=[[a],[b]]\rangle \\
&\xrightarrow{<^-} \langle \mathsf{b}=0, \mathsf{s}=[[a],[b]]\rangle \xrightarrow{>^-_a} \langle \mathsf{b}=0, \mathsf{s}=[[\,],[b]]\ \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad \xrightarrow{\}^+} \langle \mathsf{b}=0, \mathsf{s}=[[c],[b]]\rangle
\end{aligned}$$

However, this specification is not quiescently consistent with any stack execution: There is a quiescent point after each of the first two pushes; therefore it is impossible to pop a before b. This execution is possible even when there are several pushes beforehand. □

In the case of the N-`Stack`, a simple *local* constraint can be imposed in order to establish quiescent consistency: intuitively, we require that no pop *overtakes* a push on the same stack `s[i]`. In the full paper, we show that the stack is actually *QQC* under this constraint, and therefore quiescently consistent. We also prove that the elimination-tree stacks of Shavit and Touitou [16] are QQC. The increment-only counters of [3] are also QQC; the proofs for the tree-based increment-only counter follow the structure of the proofs for the elimination-tree stacks. (We have not found a *local* constraint under which the increment-decrement counter is quiescently consistent.) Our correctness result is much stronger than that of [16], which only proves *weak* quiescent consistency.

The preliminary version of Shavit and Touitou's paper [15] suggests an upcoming definition *ε-linearizability*, "a variant of linearizability that captures the notion of 'almostness' by allowing a certain fraction of concurrent operations to be out-of-order." This thread was picked up by Afek, Korland, and Yanovsky[1] and improved by Henzinger, Kirsch, Payer, Sezgin, and Sokolova[9]. As defined in [9], the idea is to define a cost metric on relaxations of strings and to bound the relaxation cost for the specification trace that matches an execution. This relaxation-based approach has been used to validate several novel concurrent data structures [1], [7]. With the exception of the increment-only counter validated in [1], all of these data structures intentionally violate quiescent consistency. In Section 4, we show that this approach in incomparable to QQC.

With QQC, the maximal degradation depends upon the amount of concurrent access, whereas in the relaxation-based approach it does not. Thus, QQC "degrades gracefully" as concurrency increases. In particular, a QQC data structure that is accessed sequentially will exactly obey the sequential specification, whereas a data structure validated against the relaxation-based approach may not.

In the rest of the paper, we formalize QQC and study its properties. Our contributions are as follows.

- We define linearizability (Section 2), quiescent consistency (Section 3) and QQC (Section 4) in terms of partial orders over events with duration. As in Example 1.1, the definitions are given in terms of the order from response to invocation.
- For sequential specifications, we provide alternative characterizations of linearizability, quiescent consistency and QQC in terms of the number of invocations that precede a response. For linearizability, this approach can be found in [4].
- We provide an alternative characterization of QQC in terms of a proxy that controls access to the underlying sequential data structure. The proxy adds a form of *speculation* to the flat combining technique of Hendler, Incze, Shavit, and Tzafrir[8]. This characterization can be seen as a language generator, rather than an accepter.
- Like linearizability and quiescent consistency [11], QQC is non-blocking and compositional. Like quiescent consistency and unlike linearizability, a QQC execution may not respect program order, and therefore QQC is incomparable to sequential consistency [12]. We prove that QQC is compositional for sequential specifications, in the sense of Herlihy and Wing[10].
- We show that QQC is useful for reasoning about data structures in the literature. In the full paper, we prove that the elimination tree stacks of Shavit and Touitou[16] are QQC, as long as no pop overtakes a push on the same stack.

2 Linearizability

A *trace* is a labelled partial order with polarity and bracketing. We use $?$ and $!$ to denote polarities. The polarity indicates whether an event in the partial order is a call/input ($?$) or a return/output ($!$). Bracketing matches each return with the particular call that precedes it. Let *p*–*t* range over traces and let a, b range over *names*, which form the carrier set of the partial order. We introduce notation over traces as needed.

Intuitively, linearizability requires that the response-to-invocation order in an execution be respected by a specification trace. To show that s'' is linearizable, it suffices to do the following

– Choose a specification trace t.
– Choose an *extension* s' of s'' that closes the open calls in s''. We say that s' *extends* s'' if (1) if s'' is a prefix of s', and (2) all of the new events in $s' - s''$ are ordered after all events of *opposite polarity* in s'' (that is, calls after returns and returns after calls). Let $\mathsf{extensions}(s'')$ be the set of extensions of s''.
– Choose a renaming $s =_{\alpha} s'$ such that $s =_{\pi} t$. Here $=_{\alpha}$ denotes equivalence up to renaming and $=_{\pi}$ denotes equivalence up to permutation. This establishes that s' is a permutation of t. The names are witness to the permutation.
– Show that for every response $a^{!}$ and invocation $b^{?}$, if $a^{!}$ precedes $b^{?}$ in s ($a^{!} \Rightarrow_{s} b^{?}$), then the same must be true in t ($a^{!} \Rightarrow_{t} b^{?}$).

This definition differs from the traditional one in several small details, enumerated in the full paper. In particular, we allow $s' \in \mathsf{extensions}(s'')$ to include calls that are not in s'', in addition to returns. We can refactor the definition slightly to pull it into the shape used to define quiescent consistency and QQC.

Definition 2.1. For traces s, t, we write $s \sqsubseteq_{\mathsf{lin}} t$ if $s =_{\pi} t$ and for every prefix $p \leq_{\mathsf{pre}} s$

$$\forall a^{!} \in p.\ \forall b^{?} \in s - p.\ (a^{!} \Rightarrow_{s} b^{?}) \text{ implies } (a^{!} \Rightarrow_{t} b^{?}).$$

Then $(s'' \underset{\sim}{\sqsubset}_{\mathsf{lin}} t) \triangleq (\exists s' \in \mathsf{extensions}(s'').\ \exists s =_{\alpha} s'.\ s \sqsubseteq_{\mathsf{lin}} t)$,
and $(S \underset{\sim}{\sqsubset}_{\mathsf{lin}} T) \triangleq (\forall s'' \in S.\ \exists t \in T.\ s'' \underset{\sim}{\sqsubset}_{\mathsf{lin}} t)$. □

This characterization of linearizability requires that we look at every way to *cut* the trace s into a prefix p and suffix $s - p$. We then look at the return events in p and the call events in $s - p$ and ensure that the order of events *crossing the cut* is respected in t. The definitions are equivalent since we quantify over all possible cuts.

Consider the counter specification from Example 1.1: $[^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ (^{+}\ )^{+}_{2}$. The trace $\{^{+}\ [^{+}\ \}^{+}_{1}\ (^{+}\ ]^{+}_{0}\ )^{+}_{2}$ is linearizable. The interesting cut is $\{^{+}\ [^{+}\ \}^{+}_{1}$ which requires only that $\{^{+}\ \}^{+}_{1}$ precede $(^{+}\ )^{+}_{2}$ in the specification. By the same reasoning, $\{^{+}\ (^{+}\ \}^{+}_{1}\ [^{+}\ )^{+}_{2}\ ]^{+}_{0}$, is not linearizable, since it requires that $\{^{+}\ \}^{+}_{1}$ precede $[^{+}\ ]^{+}_{0}$.

Given a sequential specification, a trace is linearizable if every return is preceded by the calls that come before it in specification order. This holds for *operational* traces, in which all events of opposite polarity are ordered. Operational traces correspond to those generated by a standard interleaving semantics. Define $s \leq_{\pi} t$ to mean that s is a subtrace of a permutation of t: $(s \leq_{\pi} t) \triangleq (\exists s'.\ s \subseteq s' =_{\pi} t)$.

Theorem 2.2. Let t be a sequential trace with name order $(a_1^{?}, a_1^{!}, a_2^{?}, a_2^{!}, \ldots, a_n^{?}, a_n^{!})$. *Let s be an operational trace such that* $s \leq_{\pi} t$. *Then*

$$s \underset{\sim}{\sqsubset}_{\mathsf{lin}} t \quad \textit{iff} \quad \forall a_j^{!} \in s.\ \{a_1^{?}, \ldots, a_j^{?}\} \subseteq \{a_i^{?} \mid a_i^{?} \Rightarrow_{s} a_j^{!}\}$$ □

3 Quiescent Consistency

Let $\mathsf{open}(s)$ be the set of calls in s that have no matching return. We say that trace s is *quiescent* if $\mathsf{open}(s) = \emptyset$. This notion of quiescence does not require that there be no active thread, but only that there be no open calls. Thus, this notion of quiescence is compatible with libraries that maintain their own thread pools.

The definition of quiescent consistency is similar to Definition 2.1 of linearizability. The difference lies in the quantifier for the prefix p: Whereas linearizability quantifies over *every* prefix, quiescent consistency only quantifies over *quiescent* prefixes.

Definition 3.1. We write $s \sqsubseteq_{\mathsf{qc}} t$ if $s =_\pi t$ and for any *quiescent* prefix $p \leq_{\mathsf{pre}} s$

$$\forall a^{!} \in p.\ \forall b^{?} \in s - p.\ (a^{!} \Rightarrow_s b^{?}) \text{ implies } (a^{!} \Rightarrow_t b^{?}). \qquad \square$$

$(\underset{\sim}{\sqsubset}_{\mathsf{qc}})$ is defined similarly to $(\underset{\sim}{\sqsubset}_{\mathsf{lin}})$. Again let us revisit the counter specification from Example 1.1: $[^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ (^{+}\ )^{+}_{2}$. This notion of quiescent consistency places some constraints on the system even when it has no nontrivial quiescent points. For example, the execution $[^{+}\ \{^{+}\ (^{+}\ )^{+}_{3}\ \}^{+}_{1}\ ]^{+}_{0}$ is not quiescently consistent with the given specification, since it is not a permutation. If one extends the execution to $[^{+}\ \{^{+}\ (^{+}\ )^{+}_{3}\ \}^{+}_{1}\ ]^{+}_{0}\ <^{+}\ >^{+}_{2}$ and attempts to matches it against the specification $[^{+}\ ]^{+}_{0}\ \{^{+}\ \}^{+}_{1}\ <^{+}\ >^{+}_{2}\ (^{+}\ )^{+}_{3}$, quiescent consistency continues to fail: In the quiescent prefix $[^{+}\ \{^{+}\ (^{+}\ )^{+}_{3}\ \}^{+}_{1}\ ]^{+}_{0}$, the order across the cut from $)^{+}_{3}$ to $<^{+}$ is not preserved in the specification.

For linearizability, only responses need be included in the extensions of a trace. The same does not hold for quiescent consistency. For example, since $(^{+}\ \{^{+}\ \}^{+}_{1}\ [^{+}\ ]^{+}_{0}\ )^{+}_{2}$ is quiescently consistent, its prefix $(^{+}\ \{^{+}\ \}^{+}_{1}$ should also be quiescently consistent. However, there is no specification trace that can be matched that does not include $[^{+}\ ]^{+}_{0}$. Therefore, it does not suffice merely to close the open call by adding $)^{+}_{2}$; we must also include $[^{+}$ and $]^{+}_{0}$.

We now give a counting characterization of quiescent consistency. Define $u \Mapsto_s v$ to mean that $u \Rightarrow_s v$ and there is no quiescent cut that separates u and v.

Theorem 3.2. Let t be a sequential trace with name order $(a^{?}_{1}, a^{!}_{1}, a^{?}_{2}, a^{!}_{2}, \ldots, a^{?}_{n}, a^{!}_{n})$. Let s be an operational trace such that $s \leq_\pi t$. Then

$$s \underset{\sim}{\sqsubset}_{\mathsf{qc}} t \quad \textit{iff} \quad \forall a^{!}_{j} \in s.\ \left|\{a^{?}_{1}, \ldots, a^{?}_{j}\}\right| \leq \left|\{a^{?}_{i} \mid a^{?}_{i} \Rightarrow_s a^{!}_{j}\} \cup \{a^{?}_{i} \mid a^{!}_{j} \Mapsto_s a^{?}_{i}\}\right| \qquad \square$$

If $a^{!}_{j}$, the j^{th} return in t, occurs in s, then there must be at least j calls contained in two sets: (1) the calls that precede $a^{!}_{j}$ in s, and (2) the calls that follow $a^{!}_{j}$ in s but are "quiescently concurrent" — that is, not separated by a quiescent point.

4 Quantitative Quiescent Consistency

We provide three characterizations of QQC and prove their equivalence. (1) Definition 4.1 defines QQC in the style that we have defined linearizability and quiescent consistency, from response to invocation. (2) Theorem 4.3 provides a *counting characterization* of QQC, which requires that if a response matches the i^{th} method call in the specification, then it must be preceded by at least i invocations. (3) Theorem 4.4 provides an operational characterization of QQC as a proxy between the concurrent world and an underlying sequential data structure.

To develop some intuition for the what is allowed by QQC, we give some examples using the 2-`Counter` from the introduction. First we note that the capability given by an open call can be used repeatedly, as in $(^{+}\ [^{+}\ ]^{+}_{1}\ \{^{+}\ \}^{+}_{0}\ [^{+}\ ]^{+}_{3}\ \{^{+}\ \}^{+}_{2}\ [^{+}\ ]^{+}_{5}\ \{^{+}\ \}^{+}_{4}\ )^{+}_{6}$. The open call $(^{+}$ enables the inversion of $\{^{+}\ \}^{+}_{0}$ with $[^{+}\ ]^{+}_{1}$ and also of $\{^{+}\ \}^{+}_{2}$ with $[^{+}\ ]^{+}_{3}$.

Alternatively, multiple open calls may be accumulated to create an trace with events that are arbitrarily far off, as in $(^{+}\ [^{+}\ ]^{+}_{1}\ (^{+}\ [^{+}\ ]^{+}_{3}\ (^{+}\ [^{+}\ ]^{+}_{5}\ (^{+}\ [^{+}\ ]^{+}_{7}\ [^{+}\ ]^{+}_{0}\ )^{+}_{2}\ )^{+}_{4}\ )^{+}_{6}\ )^{+}_{8}$. Note that $[^{+}\ ]^{+}_{0}$ *follows* $[^{+}\ ]^{+}_{7}$ in this execution! It is worth emphasizing that the order between these actions is observable to the outside: a single thread can call `getAndIncrement` and get 7, then subsequently call `getAndIncrement` and get 0. Such behaviors are a hallmark of nonlinearizable data structures. In general, an N-`Counter` can give results that are $k \times N$ off of the expected value, where k is the maximum number of open calls and N is the width of the counter. There is no way to bound the behavior of this counter, as in [9], without also bounding the amount of concurrency, as in [1].

It is also possible for open calls to overlap in nontrivial ways. The trace $(^{+}\ [^{+}\ ]^{+}_{1}\ \{^{+}\ [^{+}\ ]^{+}_{0}\ )^{+}_{3}\ (^{+}\ )^{+}_{2}\ \}^{+}_{4}$ is QQC. Here, the first $(^{+}$ justifies the out-of-order execution of $[^{+}\ ]^{+}_{1}$ and $[^{+}\ ]^{+}_{0}$. The subsequent $\{^{+}$ justifies an inversion of the previous justifier, namely $(^{+}\ )^{+}_{3}$ and $(^{+}\ )^{+}_{2}$. A similar example is $\{^{+}\ (^{+}\ )^{+}_{1}\ (^{+}\ [^{+}\ ]^{+}_{0}\ )^{+}_{3}\ [^{+}\ ]^{+}_{2}\ \}^{+}_{4}$.

Finally, we note that the stack execution $\{^{+}_{c}\ [^{-}\ ]^{-}_{a}\ (^{+}_{a}\ )^{+}\ \}^{+}$ is QQC with respect to the specification $(^{+}_{a}\ )^{+}\ [^{-}\ ]^{-}_{a}\ \{^{+}_{c}\ \}^{+}$. This follows from exactly the kind of reasoning that we have done for the counter. For the counter this simply means that we are seeing an integer value early, but for a stack holding pointers, it means that we can potentially see a pointer before it has been allocated! To prevent such executions, causality can be specified as a relation from calls to returns, consistent with specification order: A trace is *causal* if it respects the specified causality relation. We have elided causality from the definition of QQC because it is orthogonal and can be enforced independently.

Linearizability requires that for *every* cut, *all* response-to-invocation order crossing the cut must be respected in the specification. Quiescent consistency limits attention to *quiescent* cuts. QQC restores the quantification over every cut, but relaxes the requirement to match all response-to-invocation order crossing the cut. When checking response-to-invocation pairs across the cut, QQC allows some invocations to be ignored. How many?

One constraint comes from our desire to refine quiescent consistency. For quiescent cuts, we cannot drop any invocations, since quiescent consistency does not. As a first attempt at a definition, we may take the number of dropped invocations at any cut to be bounded by $\left|\mathsf{open}(p)\right|$. This criterion would allow both of the traces $(^{+}\ \{^{+}\ \}^{+}_{1}\ [^{+}\ ]^{+}_{0}\ )^{+}_{2}$ and $[^{+}\ (^{+}\ )^{+}_{2}\ \{^{+}\ \}^{+}_{1}\ ]^{+}_{0}$ in Example 1.1. In each case, the interesting cut splits the trace in half, with one open call and one completed. In the first trace, we can ignore $[^{+}$ in the suffix, and in the second trace, we can ignore $\{^{+}$ in the suffix; thus, both are allowed. However, in the second trace, the first call completed is two steps in the future, even though there is only one concurrent action. In the first trace this does not happen. The difference can be seen by looking not only at the number of open calls, but also at *which* calls are open. In the first trace we have $(^{+}$ before $\}^{+}_{1}$, and in the second, we have $[^{+}$ before $)^{+}_{2}$. We say that $(^{+}$ is *early* for $\}^{+}_{1}$, since it does not precede $\}^{+}_{1}$ in the specification, whereas $[^{+}$ is not early for $)^{+}_{2}$, since it *does* precede $)^{+}_{2}$. We restrict our attention to calls that are both open and early with respect to the response of interest.

Given a specification t and a response $a^{!} \in t$, none of the actions in the t-down-closure of $a^{!}$ could possibly be early for $a^{!}$; any other action could be. Thus, the actions in $\mathsf{open}(p) - (\downarrow_t a^{!})$ are both open and early for $a^{!}$. This leads us to the following definition. (In the full paper, we show that for sequential specifications, we can swap the quantifiers $(\exists r)$ and $(\forall a^{!})$, pulling out the existential.)

Definition 4.1. We write $s \sqsubseteq_{\mathsf{qqc}} t$ if $s =_{\pi} t$ and for any prefix $p \leq_{\mathsf{pre}} s$

$$\begin{aligned}&\forall a^{!} \in p.\ \exists r \subseteq s.\ \big|r\big| \leq \big|\mathsf{open}(p) - (\downarrow_t a^{!})\big|.\\ &\qquad \forall b^{?} \in ((s-p)-r).\ (a^{!} \Rightarrow_s b^{?}) \text{ implies } (a^{!} \Rightarrow_t b^{?}).\end{aligned}$$ □

As before, $(\precsim_{\mathsf{qqc}})$ is defined by analogy to $(\precsim_{\mathsf{lin}})$.

Theorem 4.2. $(\precsim_{\mathsf{lin}}) \subset (\precsim_{\mathsf{qqc}}) \subset (\precsim_{\mathsf{qc}})$ □

Given the subtlety of Definition 4.1, it may be surprising that QQC has the following simple characterization for sequential specifications.

Theorem 4.3. Let t be a sequential trace with name order $(a_1^{?}, a_1^{!}, a_2^{?}, a_2^{!}, \ldots, a_n^{?}, a_n^{!})$. Let s be an operational trace such that $s \leq_{\pi} t$. Then

$$s \precsim_{\mathsf{qqc}} t \quad \mathit{iff} \quad \forall a_j^{!} \in s.\ \big|\{a_1^{?}, \ldots, a_j^{?}\}\big| \leq \big|\{a_i^{?} \mid a_i^{?} \Rightarrow_s a_j^{!}\}\big|$$ □

This characterization provides a simple method for calculating whether a trace is QQC. For example, the trace $\{^{+}\ (^{+}\ )_1^{+}\ (^{+}\ [^{+}\ ]_0^{+}\ )_3^{+}\ [^{+}\ ]_2^{+}\ \}_4^{+}$ is QQC since $)_1^{+}$ is preceded by two calls, $]_0^{+}$, $)_3^{+}$ by four, and $]_2^{+}$, $\}_4^{+}$ by five. The trace $\{^{+}\ (^{+}\ )_1^{+}\ (^{+}\ )_3^{+}\ [^{+}\ ]_0^{+}\ [^{+}\ ]_2^{+}\ \}_4^{+}$ is not QQC since $)_3^{+}$ is only preceded by three calls, yet it is the fourth call in the specification.

Our third characterization of QQC describes how QQC affects an arbitrary sequential data structure, using a *proxy* that generates QQC traces from an underlying sequential implementation. This characterization of QQC incorporates *speculation* into flat combining [8]. We push the obligation to predict the future into the underlying sequential object, with must conform to the following interface.

```
interface Object {
  method run(i:Invocation):Response;
  method predict():Invocation;  }
```

The `run` method passes invocations to the underlying sequential structure and returns the appropriate response. The `predict` method is an oracle that guesses the invocations that are to come in the future. It is the use of `predict` that makes our code speculative.

The code for the proxy is given in Figure 1. Communication between the implementation threads and the underlying `Object` is mediated by two maps. When a thread would like to interact with the `Object`, it creates a semaphore, registers it in `called` and waits. Upon awakening, the thread removes the result from `returned` and returns.

The `Object` is serviced by a single *proxy* thread which loops forever making one of two nondeterministic choices. The proxy keeps two private maps. Upon receiving an invocation in `called`, the proxy moves the invocation from `called` to `received`. Rather than executing the received invocation, the proxy asks the oracle to predict an arbitrary invocation `i` and executes that instead, placing the result in `executed`. Once a invocation is both `received` and `executed`, it may become `returned`.

At the beginning of this section, we noted that the stack execution $\{_c^{+}\ [^{-}\ ]_a^{-}\ (_a^{+}\ )^{+}\ \}^{+}$ is QQC with respect to the specification $(_a^{+}\ )^{+}\ [^{-}\ ]_a^{-}\ \{_c^{+}\ \}^{+}$. How can such a trace possibly

```
class QQCProxy<o:Object> {
   field called:ThreadSafeMultiMap<Invocation,Semaphore> = [];
   field returned:ThreadSafeMap   <Semaphore, Response>  = [];
   method run(i:Invocation):Response { // proxy for external access to o
     val m:Semaphore = [];
     called.add(i, m);
     m.wait();
     return returned.remove(m); }
   thread { // single thread to interact with o
     val received:MultiMap<Invocation,Semaphore> = [];
     val executed:MultiMap<Invocation,Response>  = [];
     repeatedly choose {
        choice if called.notEmpty() {
           received.add(called.removeAny());
           val i:Invocation = o.predict();
           val r:Response   = o.run(i);
           executed.add(i, r); }
        choice if exists i in received.keys() intersect executed.keys() {
           val m:Semaphore = received.remove(i);
           val r:Response  = executed.remove(i);
           returned.add(m, r);
           m.signal(); } } } }
```

Fig. 1. QQC Proxy

be generated? The execution of the proxy proceeds as follows. Upon receipt of $\{^{+}_{c}$, the proxy executes $(^{+}_{a}$, storing response $)^{+}$. Upon receipt of $[^{-}$, the proxy executes $[^{-}$, storing response $]^{-}_{a}$. At this point $[^{-}\ ]^{-}_{a}$ can return. Upon receipt of $(^{+}_{a}$, the proxy executes $\{^{+}_{c}$, storing response $\}^{+}$. At this point both $(^{+}_{a}\ )^{+}$ and $\{^{+}_{c}\ \}^{+}$ can return.

Such noncausal behaviors can be eliminated by requiring when a pop is executed, a corresponding push must have been received. The prior execution is invalidated since $(^{+}_{a}\ )^{+}$ is not received when $[^{-}\ ]^{-}_{a}$ returns. However, nonlinearizable behaviors are still allowed. For example $\{^{+}_{c}\ [^{+}_{a}\ ]^{+}\ (^{+}_{b}\ )^{+}\ \}^{+}\ [^{-}\ ]^{-}_{a}\ (^{-}\ )^{-}_{b}$ is generating by predicting $(^{+}_{b}\ )^{+}$.

Theorem 4.4. The concurrent proxy is sound for QQC with respect to the underlying `Object`*. It is also complete for operational traces.* □

In the full paper, we show that the elimination-tree stack of [16] and increment-only counter of [3] are QQC. The characterizations of QQC also allow us to predict the QQC behavior of other data structures, such as a queues, even if no implementation is known. The following examples, from Sezgin[13], allow a useful comparison with [9].

To see that QQC makes distinctions not found in [9], consider the two stack traces $\{^{+}_{c}\ [^{+}_{a}\ ]^{+}\ (^{+}_{b}\ )^{+}\ <^{-}\ >^{-}_{a}\ \}^{+}$ and $\{^{+}_{c}\ [^{+}_{a}\ ]^{+}\ (^{+}_{b}\ )^{+}\ \}^{+}\ <^{-}\ >^{-}_{a}$. In the framework of [9], these are both 1 out-of-order (when a is popped, at least b must be above a on the stack). However, only the first is QQC.

In the other direction, the queue execution $\{^{+}_{a}\ [^{+}_{b_1}\ ]^{+}\ [^{+}_{b_1}\ ]^{+}\ \cdots\ [^{+}_{b_n}\ ]^{+}\ (^{+}_{c}\ )^{+}\ <^{-}\ >^{-}_{c}\ \}^{+}$ is QQC with respect to the queue specification $(^{+}_{c}\ )^{+}\ [^{+}_{b_1}\ ]^{+}\ [^{+}_{b_1}\ ]^{+}\ \cdots\ [^{+}_{b_n}\ ]^{+}\ <^{-}\ >^{-}_{c}\ \{^{+}_{a}\ \}^{+}$. In the framework of [9], this would be n out-of-order because at least all b_i's should be in the queue before c is inserted into the queue; the removal of c from the queue must happen when there are n elements ahead of c in the queue.

Finally, we prove compositionality for QQC. Let $\div$ denote partial order difference.

Theorem 4.5. Let t_1 and t_2 be sequential traces.
Let s, s_1 and s_2 be operational traces such that $s_1 = s \div s_2$ and $s_2 = s \div s_1$.
For $i \in \{1,2\}$, suppose that each $s_i \sqsubseteq_{\mathsf{qqc}} t_i$.
Then there exists a sequential trace $t \in (t_1 \mid\mid\mid t_2)$ such that $s \sqsubseteq_{\mathsf{qqc}} t$.

PROOF SKETCH. Assume that the names in t_1 and t_2 are disjoint. Let the sequence of names in t_1 be $(a^?_1, a^!_1, \ldots, a^?_m, a^!_m)$ and sequence of names in t_2 be $(b^?_1, b^!_1, \ldots, b^?_n, b^!_n)$. Applying Theorem 4.3 to the supposition $s_1 \sqsubseteq_{\mathsf{lin}} t_1$, we have that $j \leq \left|\{a^?_i \mid a^?_i \Rightarrow_s a^!_j\}\right|$, and similarly $\ell \leq \left|\{b^?_k \mid b^?_k \Rightarrow_s b^!_\ell\}\right|$. It suffices to construct an interleaving $t \in (t_1 \mid\mid\mid t_2)$ such that whenever t contains a subsequence with names

$$a^?_j, a^!_j, b^?_k, b^!_k, b^?_{k+1}, b^!_{k+1}, \ldots, b^?_{k+x}, b^!_{k+x}$$

then for every $k \leq \ell \leq k+x$, we have

$$\{a^?_i \mid a^?_i \Rightarrow_s a^!_j\} \subseteq \{a^?_i \mid a^?_i \Rightarrow_s b^!_\ell\}$$

and symmetrically for subsequences $b^?_k, b^!_k, a^?_j, a^!_j, a^?_{j+1}, a^!_{j+1}, \ldots, a^?_{j+y}, a^!_{j+y}$. To demonstrate the existence of an appropriate t, it suffices to show that $\mathsf{merge}(a^?_1 a^!_1 \ldots a^?_m a^!_m, b^?_1 b^!_1 \ldots b^?_n b^!_n)$ is nonempty. By operationality, it must be the case that either (1) $a^!_j \Rightarrow_s b^!_\ell$, in which case $\{a^?_i \mid a^?_i \Rightarrow_s a^!_j\} \subseteq \{a^?_i \mid a^?_i \Rightarrow_s b^!_\ell\}$, (2) $b^!_\ell \Rightarrow_s a^!_j$, in which case $\{b^?_k \mid b^?_k \Rightarrow_s b^!_\ell\} \subseteq \{b^?_k \mid b^?_k \Rightarrow_s a^!_j\}$, or (3) $a^!_j$ and $b^!_\ell$ are unordered, in which case both conclusions hold. Therefore an appropriate t exists. $\square$

5 Conclusions

Quantitative quiescent consistency (QQC) is a correctness criterion for concurrent data structures that relaxes linearizability and refines quiescent consistency. To the best of our knowledge, it is the first such criterion to be proposed.

To show that QQC is a robust concept, we have provided three alternate characterizations: (1) in the style of linearizability, (2) counting the number of calls before a return, and (3) using speculative flat combining. We have also proven compositionality (in the style of Herlihy and Wing [10]) and, in the full paper, the correctness of data structures defined by Aspnes, Herlihy, and Shavit [3] and Shavit and Touitou [16].

In order to establish the correctness of the elimination-tree stack of [16], we had to restrict attention to traces in which no pop *overtakes* a push on the same stack. (The formalities are given in the full paper.) A related constraint appears in a footnote of [14]: "To keep things simple, pop operations should block until a matching push appears." This, however, is not strong enough to guarantee quiescent consistency as we have defined it. Our analysis provides a full account: The stack is QQC with the no-overtaking requirement and only weakly quiescently consistent without it.

There are many unanswered questions, chief among them: Is QQC useful in reasoning about client programs? Is there a verification methodology for QQC analogous to that developed for linearizability? Are there other useful data structures that can be shown to satisfy QQC?

Linearizability is, at its core, *linear*. We have defined QQC in terms of general partial orders, and yet the results reported here are stated in terms of sequential specifications. Partly we have done this so that we can relate the definition of QQC to the vast amount of existing work on linearizability. However, the general case is interesting.

Acknowledgements. Gustavo Petri participated in the early discussions motivating this work. Alexey Gotsman suggested the connection to flat combining. Ali Sezgin provided a comparison with [9]. We also thank Alan Jeffrey, Corin Pitcher and Hongseok Yang for useful discussion.

References

[1] Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: Relaxed consistency for improved concurrency. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 395–410. Springer, Heidelberg (2010)

[2] Aiello, W., Busch, C., Herlihy, M., et al.: Supporting increment and decrementoperations in balancing networks. Chicago J. Theor. Comput. Sci. (2000)

[3] Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. ACM 41(5), 1020–1048 (1994)

[4] Batty, M., Dodds, M., Gotsman, A.: Library abstraction for C/C++ concurrency. In: POPL (2013)

[5] Busch, C., Mavronicolas, M.: The strength of counting networks (abstract). In: Burns, J.E., Moses, Y. (eds.) PODC, p. 311. ACM (1996)

[6] Dwork, C., Herlihy, M., Waarts, O.: Contention in shared memory algorithms. J. ACM 44(6), 779–805 (1997)

[7] Haas, A., Lippautz, M., Henzinger, T.A., et al.: Distributed queues in shared memory. In: Conf. Computing Frontiers, p. 17. ACM (2013)

[8] Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: SPAA, pp. 355–364 (2010)

[9] Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In: POPL, pp. 317–328 (2013)

[10] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)

[11] Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)

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

[13] Sezgin, A.: Private correspondence (March 18, 2014)

[14] Shavit, N.: Data structures in the multicore age. Commun. ACM 54(3), 76–84 (2011)

[15] Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks (preliminary version). In: SPAA, pp. 54–63 (1995)

[16] Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks. Theory Comput. Syst. 30(6), 645–670 (1997)

[17] Shavit, N., Zemach, A.: Diffracting trees. ACM Trans. Comput. Syst. 14(4), 385–428 (1996)