Round-Up: Runtime Checking Quasi Linearizability of Concurrent Data Structures

Lu Zhang, Arijit Chattopadhyay, and Chao Wang
Department of ECE, Virginia Tech
Blacksburg, VA 24061, USA
{zhanglu, arijitvt, chaowang}@vt.edu

***Abstract*—We propose a new method for runtime checking of a relaxed consistency property called *quasi linearizability* for concurrent data structures. Quasi linearizability generalizes the standard notion of *linearizability* by intentionally introducing nondeterminism into the parallel computations and exploiting such nondeterminism to improve the performance. However, ensuring the quantitative aspects of this correctness condition in the low level code is a difficult task. Our method is the first fully automated method for checking quasi linearizability in the unmodified C/C++ code of concurrent data structures. It guarantees that all the reported quasi linearizability violations are real violations. We have implemented our method in a software tool based on LLVM and a concurrency testing tool called Inspect. Our experimental evaluation shows that the new method is effective in detecting quasi linearizability violations in the source code of concurrent data structures.**

I. Introduction

Concurrent data structures are the foundation of many multi-core and high performance software systems. By providing a cost effective way to reduce the memory contention and increase the scalability, they have found increasingly widespread applications ranging from embedded computing to distributed systems such as the cloud. However, implementing concurrent data structures is not an easy task due to the subtle interactions of low level concurrent operations and often astronomically many thread interleavings. In practice, even a few hundred lines of highly concurrent low level C/C++ code can pose severe challenges for testing and debugging.

Linearizability [1], [2] is the *de facto* correctness condition for implementing concurrent data structures. It requires that every interleaved execution of the methods of a concurrent object to be equivalent, in some sense, to a sequential execution. This is extremely useful as a correctness condition for application developers because, as long as their program is correct when using the standard (sequential) data structure, switching to a concurrent version of the same data structure would not change the program behavior. Although being linearizable alone does not guarantee correctness of the program, not satisfying the linearizability requirement often indicates that the implementation is buggy.

In this paper, we propose the first fully automated method for checking *standard* and *quasi* linearizability violations of concurrent data structures. Quasi linearizability [3] is a quantitative relaxation of linearizability, which has attracted a lot of attention in recent years [4], [5], [6], [7], [8]. For many highly parallel applications, the standard notion of linearizability imposes unnecessary restrictions on the implementation, thereby leading to severe performance bottlenecks. Quasi linearizability preserves the intuition of standard linearizability while providing some additional flexibility in the implementation. For example, the task queue used in the scheduler of a thread pool does not need to follow the strict FIFO order. One can use a relaxed queue that allows some tasks to be overtaken occasionally if such relaxation leads to superior performance. Similarly, concurrent data structures used for web cache need not follow the strict semantics of the standard versions, since occasionally getting the stale data is acceptable. In distributed systems, the unique id generator does not need to be a perfect counter; to avoid becoming a performance bottleneck, it is often acceptable for the ids to be out of order occasionally, as long as it happens within a bounded time frame. Quasi linearizability allows the concurrent data structures to have such occasional deviations from the standard semantics in exchange of higher performance.

While quasi linearizable concurrent data structures have tremendous performance advantages, ensuring the quantitative aspects of this correctness condition in the actual implementation is not an easy task. To the best of our knowledge, there does not yet exist any method for checking, for example, the `deq` operation of a relaxed queue is not over-taken by other `deq` operations for more than k times. Existing methods for detecting concurrency bugs focus primarily on simple bug patterns such as deadlocks, data-races, and atomicity violations, but not this type of quantitative properties.

Broadly speaking, existing methods for checking linearizability fall into three groups. The first group consists of methods based on constructing mechanical proofs [9], [10], which require significant user intervention. The second group consists of automated methods based on model checking [11], [12], [13], which work on finite state models or abstractions of the concurrent data structure. The third group consists of runtime tools that can directly check the source code, but only for standard linearizability.

Our method is the first runtime method for checking quasi linearizability in the source code of concurrent data structures. It does not require the user to provide specifications or annotate linearization points. It takes the source code of a concurrent object o, a test program P that uses o, and a quasi factor K as input, and returns either `true` or `false` as output. It guarantees to report only real linearizability violations.

We have implemented the method in a software tool called *Round-Up* based on the LLVM compiler and Inspect [14]. It can check C/C++ programs that use the POSIX threads and

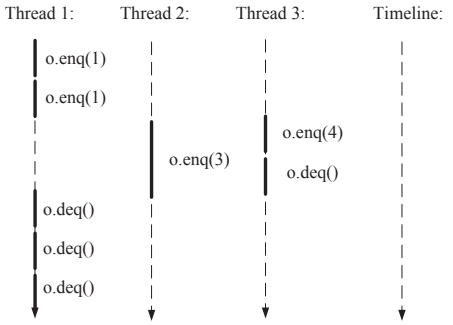

Fig. 1. A 3-threaded program that uses object o. Thread 1 starts by adding values 1 and 2 to the queue before creating two child threads. Then it waits for the child threads to terminate before removing another three data items. Here `enq(3)` runs concurrently with `enq(4)` and `deq()` in Thread 3.

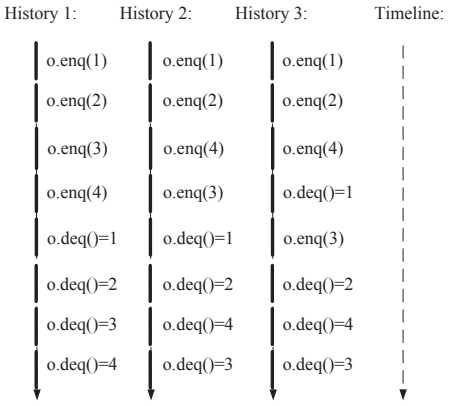

Fig. 2. The set of *legal sequential histories* generated by the program in Fig. 1. These legal sequential histories form the *sequential specification*.

GNU built-in atomic functions. Our experiments on a large set of concurrent data structure implementations show that the new method is effective in detecting both quasi linearizability violations. We have found several real implementation bugs in the *Scal* suite [15], which is an open-source package that implements some of the most recently published concurrent data structures. The bugs that we found in the *Scal* benchmarks have been confirmed by the *Scal* developers.

The remainder of this paper is organized as follows. We provide a few motivating examples in Section II and explain the main technical challenges in checking quasi linearizability. We establish notation in Section III and then present the overall algorithm in Section IV. We present the detailed algorithm for checking quasi linearizability in Section V. Our experimental results are presented in Sections VI. We review related work in Section VII, and finally give our conclusions in Section VIII.

II. Motivating Examples

In this section, we illustrate the standard and quasi linearizability properties and outline the technical challenges in checking such properties. Fig. 1 shows a multithreaded program that invokes the `enq`/`deq` methods of a queue. If Thread 2 executes `enq(3)` atomically, i.e., without interference from Thread 3, there will be three interleaved executions, all of which behave like a single-threaded execution. The sequential histories, shown in Fig. 2, satisfy the standard semantics of the queue. Therefore, we call them the *legal sequential histories*.

If the time interval of `enq(3)`, which starts at its invocation and ends at its response, overlaps with the time intervals of `enq(4)` and `deq()`, the execution is no longer sequential. In this case, the interleaved execution is called a *concurrent history*. When the implementation of the queue is linearizable, no matter how the instructions of `enq(3)` interleave with the instructions of `enq(4)` and `deq()`, the external behavior of the queue would remain the same. We say that the queue is *linearizable* if the sequence of `deq` values of any *concurrent history* matches one of the three legal sequential histories in Fig. 2. On the other hand, if the sequence of `deq` values is 3,2,1,4 in a concurrent history, we say that it has a linearizability violation, because the object no longer behaves like a FIFO queue.

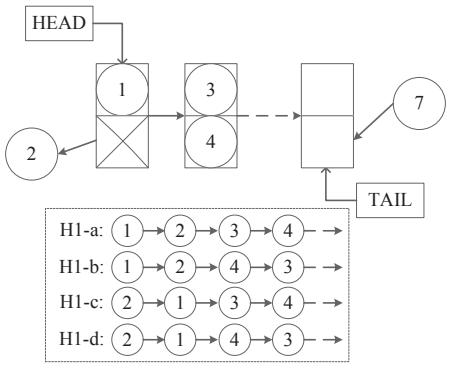

Fig. 3. An example implementation of *1-quasi linearizable* queue, where each of the linked list item is a segment that holds two data items. The first `deq` randomly returns a value from the set $\{1, 2\}$ and the second `deq` returns the remaining one. Then the third `deq` randomly returns a value from the set $\{3, 4\}$ and the fourth `deq` returns the remaining one.

However, being linearizable often means that the implementation has significant performance overhead when it is used by a large number of concurrent threads. For a quasi linearizable queue, in contrast, it is acceptable to have the `deq` values being out of order occasionally, if such relaxation of the standard semantics can help improve the performance. For example, instead of using a standard linked list to implement the queue, one may use a linked list of 2-cell segments to implement the 1-quasi linearizable queue (Fig. 3). The `deq` operation may remove any of the two data items in the head segment. By using randomization, it is possible for two threads to remove different data items from the head simultaneously without introducing memory contention.

Assume that the relaxed queue contains four values 1,2,3,4 initially. The first two `deq` operations would retrieve either 1,2 or 2,1, and the next two `deq` operations would retrieve either 3,4 or 4,3. Together, there are four possible combinations as shown in Fig. 3. Among them, $H1$-a is linearizable. The

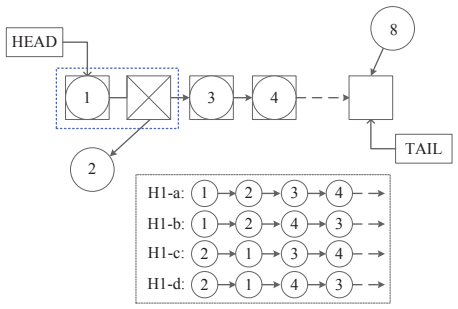

Fig. 4. An alternative implementation of *1-quasi linearizable* queue, which is based on the random-dequeued queue. The first `deq` randomly returns a value from $\{1, 2\}$ and the second `deq` returns the remaining one. Then the third `deq` randomly returns a value from the new window $\{3, 4\}$ and the fourth `deq` returns the remaining one.

other three are not linearizable, but are considered as *1-quasi linearizable*, meaning that `deq` values in these concurrent histories are out-of-order by at most one step.

However, implementing quasi linearizable data structures is a difficult task. Subtle bugs can be introduced during both the design phase and the implementation phase.

Consider an alternative way of implementing the 1-quasi linearizable queue as illustrated in Fig. 4, where the first two data items are grouped into a virtual window. A `deq` operation may retrieve any of the first 2 data items from the head based on randomization. Furthermore, only after both data items in the current window are removed, will the `deq` operation move on to retrieve data items in the next window. The resulting behavior of this implementation should be identical to that of the segmented queue.

However, a subtle bug would appear if one ignores the use of the *virtual window*. For example, if `deq` always returns one of the first two data items in the current queue, the implementation would not be 1-quasi linearizable. In this case, it is possible for some data item to be over-taken indefinitely, thereby making the data structure unsuitable for applications where a 1-quasi queue is desired. For example, if every time the `deq` operation removes *the second data item in the list*, we would get a sequence of `deq` values as follows: 2,3,4,..., where value 1 is left in the queue indefinitely.

The last example demonstrates the need for a new method that can help detect violations of such quantitative properties. Existing concurrency bug checking tools focus primarily on simple bug patterns such as deadlocks and data-races. They are not well suited for checking quantitative properties in the low level code that implements concurrent data structures. To the best of our knowledge, the method proposed in this paper is the first method for detecting quasi linearizability violations in the code of concurrent data structures.

III. Preliminaries

A. Linearizability

We follow the notation in [1], [2] to define *history* as a sequence of events, denoted $h = e_1 e_2 \dots e_n$, where each event is either a method invocation or a response of an object. When there are multiple objects, let $\rho = h|o$ denote the projection of history h to object o, which is the subsequence of events related to that object. When there are multiple threads, let $\rho|T$ denote the projection of history ρ to thread T, which is the subsequence of events of that thread. Two histories ρ and ρ' are equivalent, denoted $\rho \sim \rho'$, if and only if $\rho|T_i = \rho'|T_i$ for all thread T_i, where $i = 1, \dots, k$. Two equivalent histories have the same set of events, but the events may be arranged in different orders.

A *sequential history* is one that starts with a method invocation, and each method invocation is followed immediately by the matching response; in other words, no two method call intervals are overlapping. Otherwise, the history is called a *concurrent history*. Let $<_\rho$ be the precedence relation of events in history ρ.

Definition 1: A *linearization* of a concurrent history ρ is a sequential history ρ' such that (1) $\rho' \sim \rho$, meaning that they share the same set of events, and (2) $\forall e_i, e_j : e_i <_\rho e_j$ implies $e_i <_{\rho'} e_j$. In other words, the non-overlapping method calls in ρ retain their execution order in ρ', whereas the overlapping method calls may take effect in any order.

A *sequential specification* of object o, denoted $spec(o)$, is the set of all *legal* sequential histories – histories that conform to the semantics of the object. For example, a legal sequential history of a queue is one where all the `enq/deq` values follow the FIFO order.

Definition 2: A concurrent history ρ is linearizable with respect to a sequential specification $spec(o)$ if and only if it has a linearization ρ' such that $\rho' \in spec(o)$. In other words, as long as the concurrent history ρ can be mapped to at least one $\rho' \in spec(o)$, it is considered as linearizable.

B. Quasi Linearizability

The notion of quasi linearizability relies on the permutation distance between two sequential histories. Let $\rho' = e'_1 e'_2 \dots e'_n$ be a permutation of $\rho = e_1 e_2 \dots e_n$. Let $\Delta(\rho, \rho')$ be the distance between ρ and ρ' defined as $max_{e \in \rho}\{\ |\rho[e] - \rho'[e]|\ \}$. We use $\rho[e]$ and $\rho'[e]$ to denote the index of event e in ρ and ρ', respectively. Therefore, $\Delta(\ \rho,\ \rho'\ )$ is the maximum distance that some event in ρ has to travel to its new position in ρ'.

Quasi linearizability is often defined on a subset of the object's methods. Let $Domain(o)$ be the set of all operations of object o. Let $d \subset Domain(o)$ be a subset. Let $Powerset(Domain(o))$ be the set of all subsets of $Domain(o)$.

Definition 3: The *quasi-linearization factor* (or quasi factor) for a concurrent object o is a function $Q_o : D \to N$, where $D \subset Powerset(Domain(o))$ and N is the set of natural numbers.

For example, a queue where `enq` operations follow the FIFO order, but `deq` values may be out-of-order by at most K steps, can be specified as follows:

$$D_{\texttt{enq}} = \{\ \langle \texttt{o.enq}(x), void\rangle \mid x \in X\ \}$$
$$D_{\texttt{deq}} = \{\ \langle \texttt{o.deq}(), x\rangle \mid x \in X\ \}$$
$$Q_{\texttt{queue}}(D_{\texttt{enq}}) = 0$$
$$Q_{\texttt{queue}}(D_{\texttt{deq}}) = K$$

Definition 4: A concurrent history ρ is *quasi linearizable [3]* with respect to a sequential specification $spec(o)$ and quasi factor Q_o iff ρ has a linearization ρ' such that,

- either $\rho' \in spec(o)$, meaning that ρ is linearizable and hence is also quasi linearizable, or
- there exists a permutation ρ'' of ρ' such that
  - $\rho'' \in spec(o)$; and
  - $\Delta(\ \rho'|d,\ \rho''|d\ ) \leq Q_o(d)$ for all subset $d \in D$.

In other words, ρ' needs to be a legal sequential history, or within a bounded distance from a legal sequential history. Linearizability is subsumed by quasi linearizability with $Q_o : D \rightarrow 0$.

From now on, given a sequential history ρ', we call $\Psi = \{\ \rho'' \mid \Delta(\ \rho'|d,\ \rho''|d\ ) \leq Q_o(d)$ for all $d \in D\ \}$ the set of *quasi-permutations* of ρ'.

Quasi linearizability is compositional in that a history h is quasi linearizable if and only if subhistory $h|o$, for each object o, is quasi linearizable. This allows us to check quasi linearizability on each individual object in isolation, which reduces the computational overhead.

C. Checking (Quasi) Linearizability

There are at least three levels where one can check the (quasi) linearizability property.

- **L1:** check if a concurrent history ρ is linearizable:

  $\exists$ linearization ρ' of history ρ: $\rho' \in spec(o)$.

- **L2:** check if a concurrent program P is linearizable:

  $\forall$ concurrent history ρ of P: ρ is linearizable.

- **L3:** check if a concurrent object o is linearizable:

  $\forall$ program P that uses object o: P is linearizable.

L3 may be regarded as the full fledged verification of the concurrent object, whereas L1 and L2 may be regarded as runtime bug detection. In this paper, we focus primarily on the L1 and L2 checks. That is, given a terminating program P that uses the concurrent object o (called the test harness), we systematically generate the set of concurrent histories of P and then check if all of these concurrent histories are (quasi) linearizable. Our main contribution is to propose a new algorithm for deciding whether a concurrent history ρ is is quasi linearizable.

IV. Overall Algorithm

The overall algorithm for checking quasi linearizability consists of two phases (see Fig. 5). In Phase 1, we systematically execute the test program P together with a standard data structure to construct a sequential specification $spec(o)$, which consists of all the legal sequential histories. In Phase 2, we systematically execute the test program P together with the concurrent data structure, and for each concurrent history ρ, check whether ρ is quasi linearizable.

For data structures such as queues, stacks, and priority queues, a sequential version may serve as the golden model in Phase 1. Alternatively, the user may use a specifically configured concurrent data structure as the golden model, e.g., by setting the quasi factor of a relaxed queue to 0, which effectively turns it into a normal queue.

In Phase 1, we use a CHESS-like systematic concurrency testing tool called Inspect [14] to compute all the legal sequential histories. We have modified Inspect to automatically wrap up every method call in a lock/unlock pair. For example, method call `o.enq()` becomes `lock(lk);o.enq();unlock(lk)`, where we assign a lock lk to each object o to ensure that context switches happen only at the method call boundary. In other words, all method calls of object o are executed serially. Furthermore, Inspect can guarantee that all the possible sequential histories of this form are generated. Our new method leverages these *legal sequential histories* to construct the sequential specification $spec(o)$.

In Phase 2, we use Inspect again to compute the set of concurrent histories of the same test program. However, this time, we allow the instructions within the method bodies to interleave freely. This can be accomplished by invoking Inspect in its default mode, without adding the aforementioned lock/unlock pairs. In addition to handling the POSIX threads functions, we have extended Inspect to support the set of GNU built-in functions for atomic shared memory access, which are frequently used in implementing concurrent data structures.

Our core algorithm for checking whether a concurrent history ρ is quasi linearizable is invoked in Phase 2.

- For each concurrent history ρ, we compute the set Φ of *linearizations* of ρ (see Definition 1). If any $\rho' \in \Phi$ matches a legal sequential history in $spec(o)$, by definition, ρ is linearizable and also quasi linearizable.
- Otherwise, for each linearization $\rho' \in \Phi$, we compute the set Ψ of *quasi-permutations* of ρ' with respect to the quasi factor (see Definition 4), which defines the distance between ρ' and each $\rho'' \in \Psi$.
  - If there exists a quasi permutation ρ'' such that $\rho'' \in spec(o)$, then ρ is quasi linearizable.
  - Otherwise, ρ is not quasi linearizable and hence is not linearizable.

The pseudo code for checking quasi linearizability is shown in Algorithm 1, which takes a concurrent history ρ and a quasi factor K as input and returns either TRUE (quasi linearizable) or FALSE (not quasi linearizable). The main challenge is to generate the set Φ of linearizations of the given history ρ and the set Ψ of quasi permutations of each $\rho' \in \Phi$. The first step, which is straightforward, will be explained in this section. The second step, which is significantly more involved, will be explained in the next section.

We now explain the detailed algorithm for computing the set Φ of linearizations for the given history ρ. The computation is carried out by Subroutine `compute_linearizations`(ρ). Let history $\rho_0 = \varphi\ \texttt{inv}_1\ \texttt{inv}_2\ \phi\ \texttt{resp}_1\ \psi\ \texttt{resp}_2\ \ldots$ where φ, ϕ and ψ are arbitrary subsequences and $\texttt{inv}_1, \texttt{inv}_2$ are the invocation events of the first two overlapping method calls. We will replace ρ_0 in Φ with the new histories ρ_1 and ρ_2. In other words, for any two method call pairs $(\texttt{inv}_i, \texttt{resp}_i)$ and $(\texttt{inv}_j, \texttt{resp}_j)$ in ρ, if they do not overlap, meaning that either $\texttt{resp}_i <_\rho \texttt{inv}_j$ or $\texttt{resp}_j <_\rho \texttt{inv}_i$, we will keep this execution order. In they overlap, we will generate two new

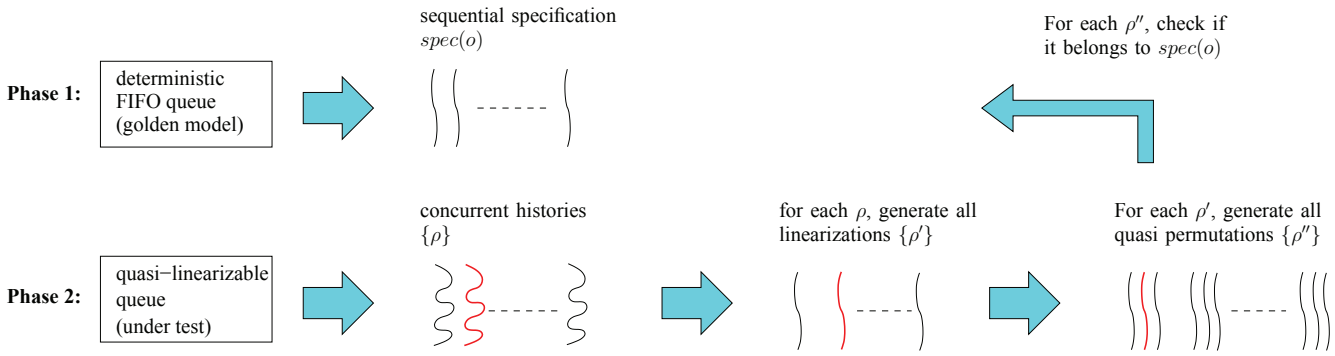

Fig. 5. The overall flow of our new quasi linearizability checking algorithm.

Algorithm 1 Checking the quasi linearizability of the concurrent history ρ with respect to the quasi factor K.

```
1: check_quasi_linearizability ( ρ, K )
2: {
3:     Φ ← compute_linearizations( ρ );
4:     for each ( ρ′ ∈ Φ ) {
5:         if ( ρ′ ∈ spec(o) ) return TRUE;
6:         Ψ ← compute_quasi_permutations( ρ′, K );
7:         for each ( ρ″ ∈ Ψ ) {
8:             if ( ρ″ ∈ spec(o) ) return TRUE;
9:         }
10:    }
11:    return FALSE;
12: }
13: compute_linearizations ( ρ )
14: {
15:    Φ ← {ρ};
16:    while ( ∃ a concurrent history ρ0 ∈ Φ ) {
17:        Let ρ0 = φ inv1 inv2 φ resp1 ψ resp2 ...;
18:        ρ1 ← φ inv1 resp1 inv2 φ ψ resp2 ...;
19:        ρ2 ← φ inv2 resp2 inv1 φ resp1 ψ ...;
20:        Φ ← Φ ∪ {ρ1, ρ2} \ {ρ0};
21:    }
22:    return Φ;
23: }
24: compute_quasi_permutations ( ρ′, K )
25: {
26:    Ψ ← { };
27:    state_stack ← first_run( ρ′, K );
28:    while ( TRUE ) {
29:        ρ″ ← backtrack_run ( state_stack, ρ′ );
30:        if ( ρ″ = = null ) break;
31:        Ψ ← Ψ ∪ {ρ″};
32:    }
33:    return Ψ;
34: }
```

histories, where one has $\texttt{resp}_i <_\rho \texttt{inv}_j$ and the other has $\texttt{resp}_j <_\rho \texttt{inv}_i$.

Example. Consider the history in Fig. 6 (left). The first two overlapping calls start with $\texttt{inv}_1$ and $\texttt{inv}_2$, respectively.

- First, we construct a new history where $(\texttt{inv}_1, \texttt{resp}_1)$ is moved ahead of $(\texttt{inv}_2, \texttt{resp}_2)$. This is straightforward because, by the time we identify $\texttt{inv}_1$ and $\texttt{inv}_2$, we can continue to traverse the event sequence to find $\texttt{resp}_1$ in ρ_0 and then move it ahead of event $\texttt{inv}_2$. Since the resulting `History 1` still has overlapping method calls, we repeat the process in the next iteration.
- Second, we construct a new history by moving $(\texttt{inv}_2, \texttt{resp}_2)$ ahead of $(\texttt{inv}_1, \texttt{resp}_1)$. This is a little more involved because there can be many other method calls of Thread T_1 that are executed between $\texttt{inv}_2$ and $\texttt{resp}_2$.

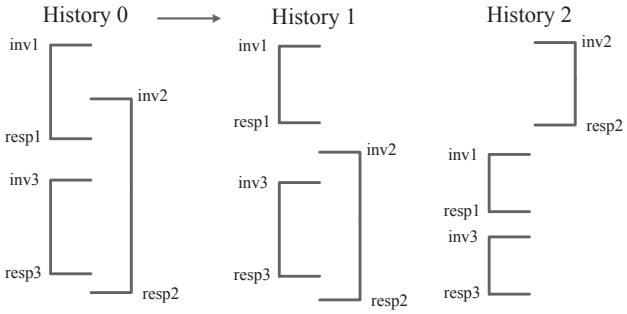

Fig. 6. Example: Computing the *linearizations* of the given concurrent history by repeatedly sequentializing the first two overlapping method calls denoted by `(inv1,resp1)` and `(inv2,resp2)`.

We take all these events between $\texttt{inv}_1$ and $\texttt{resp}_2$, and move them after $\texttt{resp}_2$. In this example, the new history is `History 2`.

According to Definition 1, when at least one of the linearizations $\rho' \in \Phi$ is a legal sequential history, ρ is linearizable, which means that it is also quasi linearizable. Otherwise, ρ is not linearizable (but may still be quasi linearizable).

V. Checking for Quasi Linearizability

To check whether history $\rho' \in \Phi$ is still quasi linearizable, we need to invoke Subroutine `compute_quasi_permutations`(ρ', K). As shown in Algorithm 1, the subroutine consists of two steps. In the first step, `first_run` is invoked to construct a doubly linked list to hold the sequence of states connected by events in ρ', denoted `state_stack`: $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \ldots s_n \xrightarrow{e_n}$. Each state s_i, where $i = 1, \ldots, n$, represents an abstract state of the object o. Subroutine `first_run` also fills up the fields of each state with the information needed later to generate the quasi permutations. In the second step, we generate quasi permutations of $\rho' \in \Psi$, one at a time, by calling `backtrack_run`.

A. Example: Constructing Quasi Permutations

We generate the quasi permutations by reshuffling the events in ρ' to form new histories. More specifically, we compute

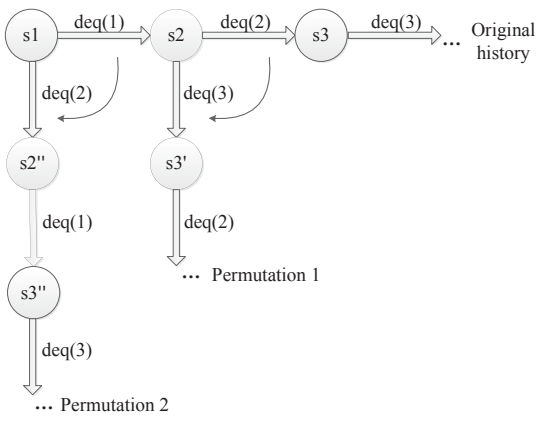

Fig. 7. An example search tree for generating all 1-quasi permutations of the input sequence `deq(1);deq(2);deq(3)`.

all possible permutations of ρ', denoted $\{\rho''\}$, such that the distance between ρ' and ρ'' are bounded by the quasi factor K. Our method for constructing the quasi permutations follows the *strict out-of-order* semantics as defined in [3], [4]. Consider queues as the example. A *strict out-of-order* k-quasi permutation consists of two restrictions:

- **Restriction 1**: each `deq` is allowed to return a value that is at most k steps away from the head node.
- **Restriction 2**: the first data element (in head node) must be returned by one of the first k `deq` operations.

```
History 0:  deq(1) --> deq(2) --> deq(3)
                                              Res1      Res2

History 1:  deq(2) --> deq(1) --> deq(3)      ok        ok
History 2:  deq(1) --> deq(3) --> deq(2)      ok        ok
History 3:  deq(3) --> deq(1) --> deq(2)      NO        ok
History 4:  deq(2) --> deq(3) --> deq(1)      ok        NO
History 5:  deq(3) --> deq(2) --> deq(1)      NO        NO
```

To illustrate the *strict out-of-order* definition, consider the 1-quasi queue above. Assume that the input history ρ' is `deq()=1,deq()=2,deq()=3`. The history can be arbitrarily reshuffled into five additional histories, of which only History 1 and History 2 satisfy the above two restrictions. They are the desired quasi permutations of ρ' whereas the others are not. In particular, History 3 violates Restriction 1 because the first `deq` returns the value that is two steps away from the head. History 4 violates Restriction 2 because the head value is returned by the third `deq` operation, which is too late. History 5 violates both restrictions.

We compute the quasi permutations using a depth-first search (DFS) of the abstract states. For the above example, this process is illustrated in Fig. 7, where the initial run is assumed to be $s_1 \xrightarrow{\texttt{deq(1)}} s_2 \xrightarrow{\texttt{deq(2)}} s_3 \xrightarrow{\texttt{deq(3)}}$.

- In the first run, we construct the state stack that holds the initial history. Then we find the last backtrack state, which is state s_2, and execute `deq(3)` instead of `deq(2)`. This leads to the second run $s_1 \xrightarrow{\texttt{deq(1)}} s_2 \xrightarrow{\texttt{deq(3)}} s'_3 \xrightarrow{\texttt{deq(2)}}$.
- In the second run, we again find the last backtrack state, which is s_1, and execute `deq(2)` instead of `deq(1)`. This leads to the third run $s_1 \xrightarrow{\texttt{deq(2)}} s''_2 \xrightarrow{\texttt{deq(1)}} s''_3 \xrightarrow{\texttt{deq(3)}}$.
- In the third run, we can no longer find any backtrack state. Therefore, the procedure terminates. We cannot generate a new run by choosing `deq(3)` in state s_1, because it would violate Restriction 1. We cannot generate a new run by choosing `deq(3)` in state s''_2 either, because it would violate Restriction 2.

B. Elementary Data Structures

To enforce the restrictions imposed by the *strict out-of-order* semantics, we need to add some fields into each state. In particular, we add an *enabled* field into each state to help enforce Restriction 1, and we add a *lateness* attribute into each enabled event to enforce Restriction 2.

state_stack: We store the sequence of states of the current run in a doubly linked list called state_stack. Executing a method call event moves the object from one state to another state. Each state s has the following fields:

- s.enabled is the set of events that can be executed at s;
- s.select is the event executed by the current history;
- s.done is the set of events executed at s by some previously explored permutations in the backtrack search;
- s.newly_enabled is the set of events that become enabled for the first time along the given history ρ'. The field is initialized by the first run, and is used to compute the s.enabled field in the subsequent runs.

Example 1: `s.newly_enabled`. The initial state has at most $(K+1)$ events in its newly_enabled field, where K is the quasi factor. Every other state has at most one event in this newly_enabled field. For the given history `deq(1);deq(2);deq(3)` and quasi factor 1, we have

s_1.newly_enabled={deq(1),deq(2)}	$\geq$1 events in the initial state
s_2.newly_enabled={deq(3)}	at most one event
s_3.newly_enabled={ }	at most one event

In other words, each event will appear in the newly_enabled field of the state that is precisely K steps ahead of its original state in ρ'. We will enforce Restriction 1 with the help of the newly_enabled field.

Example 2: `s.enabled` and `s.done`. For the above example,

s_1.enabled={deq(1),deq(2)}	s_1.done={deq(1)}
s_2.enabled={deq(2),deq(3)}	s_2.done={deq(2)}
s_3.enabled={deq(3)}	s_3.done={deq(3)}

Both `deq(1)` and `deq(2)` are in s_1.enabled, but only `deq(1)` is in s_1.done because it is executed in the current run. Since the set (s.enabled$\backslash s$.done) is not empty for both s_1 and s_2, we have two backtrack states. After backtracking to s_2 and executing `deq(3)`, we create a new permutation `deq(1);deq(3);deq(2)`. Similarly, after backtracking to s_1 and executing `deq(2)`, we create a new permutation `deq(2);deq(1);deq(3)`.

For permutation `deq(2);deq(1);deq(3)`, the enabled and done fields will be changed to the following:

s_1.enabled={deq(1),deq(2)}	s_1.done={deq(1),deq(2)}
s''_2.enabled={deq(1),deq(3)}	s''_2.done={deq(1)}
s''_3.enabled={deq(3)}	s''_3.done={deq3}

Although (s''_2.enabled$\backslash s''_2$.done) is not empty, we cannot create the new permutation `deq(2);deq(3);deq(1)` because `deq(1)` would be out-of-order by two steps. We avoid generating such permutations by leveraging the lateness attribute that is added into every enabled event.

lateness attribute: Each event in s.enabled has a lateness attribute, indicating how many steps this event is later than its original occurrence in ρ'. It represents how many steps this event can be postponed further in the current permutation.

```
s[i-k]                lateness(e) = -k
...
s[i].select = e       lateness(e) =  0
...
s[i+k]                lateness(e) =  k
```

Example 3: Consider the example above, where event e is executed in state s_i of the given history. For k-quasi permutations, the earliest state where e may be executed is s_{i-k}, and the latest state where e may be executed is s_{i+k}. The lateness attribute of event e in state s_{i-k} is $-k$, meaning that it may be postponed for at most $k-(-k) = 2k$ steps. The lateness of e in state s_{i+k} is k, meaning that e has reached the maximum lateness and therefore must be executed in this state.

Must-select event: This brings us to the important notion of must-select event. In s.enabled, if there does not exist any event whose lateness reaches k, all the enabled events can be postponed for at least one more step. In this case, we can randomly choose an event from the set (s.enabled \s.$done$) to execute. If there exists an event in s.enabled whose lateness is k, then we must execute this event in state s.

Example 4: If we backtrack from the current history `deq(1),deq(2),deq(3)` to state s_1 and then execute `deq(2)`, event `deq(1)` will have a lateness of 1 in state s''_2, meaning that it has reached the maximum delay allowed. Therefore, it has to be executed in state s_2.

s_1.lateness={deq(1):lateness=0, deq(2):lateness=-1}
s''_2.lateness={deq(1):lateness=1, deq(3):lateness=-1}
s''_3.lateness={deq(3):lateness=0}

The initial lateness is assigned to each enabled event when the event is added to s.enabled by `first_run`. Every time an event is not selected for execution in the current state, it will be inherited by the enabled field of the subsequent state. The lateness of this event is then increased by 1.

An important observation is that, in each state, there can be at most one *must-select* event. This is because the first run ρ' is a total order of events, which gives each event a different *lateness* value – by definition, their *expiration times* are all different.

C. Algorithm: Constructing K-Quasi Permutations

The pseudo code for generating quasi permutations of history ρ' is shown in Algorithm 2. Initializing the lateness attributes of enabled events is performed by Subroutine `init_enabled_and_lateness`, which is called by `first_run`. The lateness attributes are then updated by `update_enabled_and_lateness`.

Each call to `backtrack_run` will return a new quasi permutation of ρ'. Inside this subroutine, we search for the last backtrack state s in state_stack. If such backtrack state s exists, we prepare the generation of a new permutation by resetting the fields of all subsequent states of s, while keeping their newly_enabled fields intact. Then we choose a previously unexplored event in s.enabled to execute.

The previously unexplored event in s.enabled is chosen by calling `pick_an_enabled_event`. If there exists a must-select event in s.enabled whose lateness reaches k, then it must be chosen. Otherwise, we choose an event from the set (s.enabled $\backslash s$.done) arbitrarily. We use `update_enabled_and_lateness` to fill up the events in s.enabled. For events that are inherited from the previous state's enabled set, we increase their lateness' by one. We iterate until the last state is reached. At this time, we have computed a new quasi permutation of ρ'.

D. Discussions

Our method is geared toward bug hunting. Whenever we find a concurrent history ρ that is not quasi linearizable, it is guaranteed to be a real violation. However, recall that our method implements the L1 and L2 checks but not the L3 check as defined in Section III. Therefore, even if all concurrent histories of the test program are quasi linearizable, we cannot conclude that the concurrent data structure itself is quasi linearizable.

Furthermore, when checking for quasi linearizability, our runtime checking framework has the capability of generating test programs (harness) that are *well-formed*; that is, the number of `enq` operations is equal to the number of `deq` operations. If the test program is provided by the user, then it is the user's responsibility to ensure this well-formedness. This is important because, if the test program is not well-formed, there may be *out-of-thin-air* events. Below is an example.

Thread 1	Thread 2	Hist1	Hist2	Hist3
enq(3)	enq(5)	enq(3)	enq(5)	enq(3)
enq(4)	deq()	...	...	enq(4)
...	...	...	...	enq(5)
		deq()=3	deq()=5	deq()=4

Here, the sequential specification is {`Hist1`,`Hist2`}. In both histories, either `deq()=3` or `deq()=5`. However, the `deq` value can never be 4. This is unfortunate because `Hist3` is 1-quasi linearizable but cannot match any of the two legal sequential histories (`Hist1` or `Hist2`) because it has `deq()=4`. This problem can be avoided by requiring the test program to be well-formed. For example, by adding two more `deq` calls to the end of the main thread, we can avoid the aforementioned *out-of-thin-air* events.

VI. Experiments

We have implemented our new quasi linearizability checking method in a software tool based on the LLVM platform for code instrumentation and based on *Inspect* for systematically generating interleaved executions. Our tool, called *Round-Up*, can handle unmodified C/C++ code of concurrent data structures on the Linux/PThreads platform. We have improved *Inspect* by adding the support for GNU built-in atomic functions for direct access of shared memory, since they are frequently used in the low level code for implementing concurrent data structures.

Algorithm 2 Generating K-quasi permutations for history ρ'.

```
1: first_run ( ρ', K ) {
2:     state_stack ← empty list;
3:     for each ( event ev in the sequence ρ' ) {
4:         s ← new state;
5:         state_stack.append( s );
6:         s.select ← ev;
7:         s.done ← {ev};
8:         init_enabled_and_lateness( s, ev, K );
9:     }
10:    return state_stack;
11: }
12: init_enabled_and_lateness ( s, ev, K ) {
13:    lateness ← 0;
14:    while( 1 ) {
15:        s.enabled.add( ⟨ev,lateness⟩ );
16:        if( lateness == −k || s.prev == null ) {
17:            s.newly_enabled.add( ⟨ev, lateness⟩ );
18:            break;
19:        }
20:        lateness–;
21:        s ← s.prev in state_stack;
22:    }
23: }
24: backtrack_run ( state_stack, ρ ) {
25:    Let s be the last state in state_stack such that
26:        pick_an_enabled_event( s ) ≠ null;
27:    if( such s does not exist )
28:        return null;
29:    for each( state after s in state_stack ) {
30:        reset s.select, s.done, and s.enabled,
31:        but keep s.newly_enabled;
32:    }
33:    while( s ≠ null ) {
34:        ev ←pick_an_enabled_event ( s );
35:        s.select ← ev;
36:        s.done ← {ev};
37:        s ← s.next;
38:        update_enabled_and_lateness( s );
39:    }
40:    return (sequence of selected events in state_stack);
41: }
42: pick_an_enabled_event ( s ) {
43:    if( ∃⟨ev, lateness⟩ ∈ s.enabled && lateness = k) {
44:        if( ev ∉ s.done )     // must-select event
45:            return ev;
46:        else
47:            return null;
48:    }
49:    if( ∃⟨ev, lateness⟩ ∈ s.enabled && ev ∉ s.done) )
50:        return ev;
51:    else
52:        return null;
53: }
54: update_enabled_and_lateness ( s ) {
55:    p ← s.prev;
56:    if( s or p do not exist )
57:        return;
58:    s.enabled ← { };
59:    for each( ⟨ev, lateness⟩ ∈ p.enabled && ev ∉ p.done) {
60:        s.enabled.add( ⟨ev, lateness- - ⟩ );
61:    }
62:    for each( ⟨ev, lateness⟩ ∈ s.newly_enabled ) {
63:        s.enabled.add( ⟨ev, lateness⟩ );
64:    }
65: }
```

We have conducted experiments on a set of concurrent data structures [3], [6], [8], [5], [6], [4] including both standard and quasi linearizable queues, stacks, and priority queues. For some data structures, there are several variants, each of which uses a different implementation scheme. The characteristics of these benchmark programs are shown in Table I. The first three columns list the name of the data structure, a short description, and the number of lines of code. The next two columns show whether it is linearizable and quasi linearizable. The last column provides a list of the relevant methods.

Table II shows the experimental results for checking standard linearizability. The first four columns show the statistics of the test program, including the name, the number of threads (concurrent/total), the number of method calls, and whether linearizability violations exist. The next two columns show the statistics of Phase 1, consisting of the number of sequential histories and the time for generating these sequential histories. The last three columns show the statistics of Phase 2, consisting of the number of concurrent histories (buggy/total), the total number of linearizations, and the time for checking them. In all test cases, our method was able to correctly detect the linearizability violations.

Table III shows the experimental results for checking quasi linearizability. The first four columns show the statistics of the test program. The next two columns show the statistics of Phase 1, and the last three columns show the statistics of Phase 2, consisting of the number of concurrent histories (buggy/total), the total number of quasi permutations, and the time for generating and checking them. In all test cases, we have set the quasi factor to 2.

Our method was able to detect all real (quasi) linearizability violations in fairly small test programs. This is consistent with the experience of Burckhart *et al.* [16] in evaluating their Line-Up tool for checking standard (but not quasi) linearizability. This is due to the particular application of checking the implementation of concurrent data structures. Although the number of method calls in the test program is small, the underlying low-level shared memory operations can still be many. This leads to a rich set of very subtle interactions between the low-level memory accessing instructions. In such cases, the buggy execution can be uncovered by checking a test program with only a relatively small number of threads, method calls, and context switches.

We have also conducted experiments on a set of recently released high-performance concurrent objects in the *Scal* suite [15]. Table IV shows the characteristics of these benchmark programs and Table V shows the experimental results. We have successfully detected two real linearizability violations in the *Scal* suite, one of which is a known violation whereas the other is a previously unknown programming error. In particular, *sl-queue* is a queue designed for high performance applications, but it is not thread safe and therefore is not linearizable. *k-stack*, on the other hand, is designed to be quasi linearizable. However, due to an ABA bug, the data structure is not quasi linearizable.

Our tool is able to quickly detect the linearizability violation in *sl-queue* and the quasi linearizability violation in *k-stack*. Furthermore, it generates detailed execution traces to illustrate how the violations can be reproduced during debugging. In terms of the ABA bug in *k-stack*, for example, our tool shows that the bug occurs when one thread executes the `push` operation while another thread is executing the `pop` operation concurrently. Due to erroneous thread interleaving, it is possible for the same data item to be added to the stack twice, although the *push* operation is executed only once. We have reported the bug in *k-stack* to the *Scal* developers, who have confirmed that it is indeed a bug.

It is worth pointing out that existing concurrency bug finding tools, such as data-race and atomicity violation detectors, are

TABLE I
THE STATISTICS OF THE BENCHMARK EXAMPLES.

Class	Description	LOC	Linearizable	Quasi-Lin	Methods checked
IQueue	buggy queue, deq may remove null even if not empty	154	No	NO	enq(int), deq()
Herlihy/Wing queue	correct normal queue	109	YES	YES	enq(int), deq()
Quasi Queue	correct quasi queue	464	NO	YES	enq(int), deq()
Quasi Queue b1	deq removes value more than k away from head	704	NO	NO	enq(int), deq()
Quasi Queue b2	deq removes values that have been removed before	401	NO	NO	enq(int), deq()
Quasi Queue b3	deq null even the queue is not empty	427	NO	NO	enq(int), deq()
Quasi Stack b1	pop null even if the stack is not empty	487	NO	NO	push(int), pop()
Quasi Stack b2	pop removes values move than k away from the tail	403	NO	NO	push(int), pop()
Quasi Stack	linearizable, and hence quasi linearizable	403	YES	YES	push(int), pop()
Quasi Priority Queue	implementation of quasi priority queue	508	NO	YES	enq(int, int), deqMin()
Quasi Priority Queue b2	deqMin removes value more than k away from head	537	NO	NO	enq(int, int), deqMin()

TABLE II
RESULTS OF CHECKING STANDARD LINEARIZABILITY ON CONCURRENT DATA STRUCTURES.

Test Program				Phase 1		Phase 2		
Class	threads	calls	violation	history	time (seconds)	history (buggy/total)	linearization	time (seconds)
IQueue	2/3	2*2+0	YES	3	0.1	2/6	13	0.3
Herlihy/Wing queue	2/3	2*2+0	NO	3	0.1	0/4	9	0.2
Quasi Queue	2/3	2*2+4	YES	6	0.2	16/16	61	2.5
Quasi Queue	2/3	3*3+4	YES	20	1.1	64/64	505	43.7
Quasi Queue	2/3	2*3+3	YES	10	0.4	24/32	169	8.3
Quasi Queue	2/3	3*3+2	YES	20	0.8	108/118	1033	1m23s
Quasi Queue	2/3	3*4+1	YES	35	1.6	149/198	2260	5m8s
Quasi Queue	2/3	4*4+0	YES	70	2.6	274/476	8484	37m34s
Quasi Queue b1	2/3	2*2+4	YES	6	0.3	91/91	409	17.8
Quasi Queue b2	2/3	2*2+4	YES	6	0.3	91/91	409	18.1
Quasi Queue b3	2/3	2*2+4	YES	6	0.3	141/141	653	26.9
Quasi Stack b1	2/3	2*2+4	YES	6	0.3	9/9	34	1.6
Quasi Stack b2	2/3	2*2+4	YES	6	0.3	16/16	61	2.5
Quasi Stack	2/3	2*2+4	NO	6	0.3	0/16	61	2.5
Quasi Priority Queue	2/3	2*2+4	YES	6	0.5	16/16	61	4.9
Quasi Priority Queue b2	2/3	2*2+4	YES	6	0.5	125/125	532	27.0

TABLE III
RESULTS OF CHECKING QUASI LINEARIZABILITY ON CONCURRENT DATA STRUCTURES.

Test Program				Phase 1		Phase 2		
Class	threads	calls	violation	history	time (seconds)	history (buggy/total)	permutation	time (seconds)
Quasi Queue	2/3	2*2+4	NO	6	0.2	0/16	1708	2.9
Quasi Queue	2/3	3*3+4	NO	20	1.1	0/64	73730	5m33s
Quasi Queue	2/3	2*3+3	NO	10	0.4	0/32	4732	9.6
Quasi Queue	2/3	3*3+2	NO	20	0.8	0/118	28924	1m34s
Quasi Queue	2/3	3*4+1	NO	35	1.6	0/198	63280	5m40s
Quasi Queue	2/3	4*4+0	NO	70	2.6	0/476	237552	40m56s
Quasi Queue (qfactor=3)	2/3	2*3+3	NO	10	0.4	0/32	8112	14.9
Quasi Queue (qfactor=3)	2/3	3*3+2	NO	20	0.8	0/118	49584	2m36s
Quasi Queue (qfactor=3)	2/3	3*4+1	NO	35	1.6	0/198	108480	10m15s
Quasi Queue (qfactor=3)	2/3	4*4+0	NO	70	2.6	0/476	407232	69m32s
Quasi Queue b1	2/3	2*2+4	YES	6	0.3	41/91	11452	20.1
Quasi Queue b2	2/3	2*2+4	YES	6	0.3	91/91	11452	20.2
Quasi Queue b3	2/3	2*2+4	YES	6	0.3	73/141	18284	31.0
Quasi Stack b1	2/3	2*2+4	YES	6	0.3	9/9	2108	3.5
Quasi Stack b2	2/3	2*2+4	YES	6	0.3	6/16	1708	2.8
Quasi Stack b3	2/3	2*2+4	NO	6	0.3	0/16	1708	2.8
Quasi Priority Queue	2/3	2*2+4	NO	6	0.5	0/16	1708	4.7
Quasi Priority Queue b2	2/3	2*2+4	YES	6	0.5	54/125	6384	20.0

TABLE IV
THE STATISTICS OF THE SCAL [15] BENCHMARK EXAMPLES (TOTAL LOC OF *Scal* IS 5,973).

Class	Description	LOC	Linearizable	Quasi-Lin	Methods checked
sl-queue	singly-linked list based single-threaded queue	73	NO	NO	enq, deq
t-stack	concurrent stack by R. K. Treiber	109	YES	YES	push, pop
ms-queue	concurrent queue by M. Michael and M. Scott	250	YES	YES	enq, deq
rd-queue	random dequeued queue by Y. Afek, G. Korland, and E. Yanovsky	162	NO	YES	enq, deq
bk-queue	bounded k-FIFO queue by Y. Afek, G. Korland, and E. Yanovsky	263	NO	YES	enq, deq
ubk-queue	unbounded k-FIFO queue by C.M. Kirsch, M. Lippautz, and H. Payer	259	NO	YES	enq, deq
k-stack	k-stack by T. A. Henzinger, C. M. Kirsch, H. Payer, and A. Sokolova	337	NO	NO	push, pop

TABLE V
RESULTS OF CHECKING QUASI LINEARIZABILITY FOR THE SCAL [15] BENCHMARK EXAMPLES.

Test Program				Phase 1		Phase 2		
Class	threads	calls	violation	history	time (seconds)	history (buggy/total)	permutation	time (seconds)
sl-queue (enq+deq)	2/3	1*1+10	NO	2	0.1	0/2	438	0.5
sl-queue (enq+enq)	2/3	1*1+10	YES	2	0.06	1/2	438	0.54
sl-queue (deq+deq)	2/3	1*1+10	YES	2	0.07	4/8	2190	2.29
t-stack (push+pop)	2/3	1*1+10	NO	2	0.16	0/8	2190	2.6
t-stack (push+push)	2/3	1*1+10	NO	2	0.1	0/8	2190	2.45
t-stack (pop+pop)	2/3	1*1+10	NO	2	0.12	0/8	2190	2.34
ms-queue (enq+deq)	2/3	1*1+10	NO	2	0.11	0/3	730	0.96
ms-queue (enq+enq)	2/3	1*1+10	NO	2	0.12	0/31	8906	10.78
ms-queue (deq+deq)	2/3	1*1+10	NO	2	0.13	0/12	3358	3.68
rd-queue (enq+deq)	2/3	1*1+10	NO	2	0.25	0/7	1898	2.63
rd-queue (enq+enq)	2/3	1*1+10	NO	2	0.2	0/31	8906	11.23
rd-queue (deq+deq)	2/3	1*1+10	NO	2	0.13	0/6	1606	2.04
bk-queue (enq+deq)	2/3	1*1+10	NO	2	0.23	0/1	146	0.22
bk-queue (enq+enq)	2/3	1*1+10	NO	2	0.18	0/12	3358	3.94
bk-queue (deq+deq)	2/3	1*1+10	NO	2	0.19	0/8	2190	2.74
ubk-queue (enq+deq)	2/3	1*1+10	NO	2	0.85	0/1	146	0.25
ubk-queue (enq+enq)	2/3	1*1+10	NO	2	0.65	0/12	3358	6.55
ubk-queue (deq+deq)	2/3	1*1+10	NO	2	0.28	0/8	2190	3.2
k-stack (push+pop)	2/3	1*1+10	YES	2	0.82	11/69	20002	27.35
k-stack (push+push)	2/3	1*1+10	NO	2	0.26	0/12	3358	4.86
k-stack (pop+pop)	2/3	1*1+10	NO	2	0.34	0/8	2190	3.85

not effective for checking low-level C/C++ code that implements most of the highly concurrent data structures. These bug detectors are designed primarily for checking application level code. Furthermore, they are often based on the lockset analysis and condition variable analysis. Although locks and condition variables are widely used in writing application level code, they are rarely used in implementing concurrent data structures. Synchronization in concurrent data structures may be implemented using atomic memory accesses. To the best of our knowledge, no prior method can directly check quantitative properties in such low level C/C++ code.

VII. RELATED WORK

Our new method can detect quasi linearizability violations in the code of concurrent data structures. A closely related work is a model checking based approach for formally verifying quantitative relaxations of linearizability in models of concurrent systems, which we have published recently [17]. However, the method is not designed for checking the C/C++ code. Another closely related work is Line-Up [16], which can check the code of concurrent data structures for *deterministic linearizability* but cannot check for quasi linearizability.

There exists a large body of work on verifying standard linearizability. For example, Liu et al. [11] verify standard linearizability by proving that an implementation model refines a specification model. Vechev et al. [12] use the SPIN model checker to verify linearizability in a Promela model. Cerný et al. [13] use automated abstractions together with model checking to verify linearizability properties in Java programs. There also exists work on proving linearizability by constructing mechanical proofs, often with significant manual intervention [9], [10]. However, none of these methods can check quantitative relaxations of linearizability.

There also exist runtime checking methods for other types of consistency conditions such as sequential consistency [18], quiescent consistency [19], and eventual consistency [20]. Some of these consistency conditions, in principle, may be used to ensure the correctness of concurrent data structures. However, none of these correctness conditions is as widely used as linearizability. Furthermore, they do not involve any quantitative properties.

For checking application level code, which has significantly different characteristics from the low level code that implements concurrent data structures, *serializability* and *atomicity* are the two frequently used correctness properties. There also exists a large body of work on detecting violations of these properties (e.g. [21], [22], [23] and [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43]). These bug finding methods differ from our new method in that they are checking for different types of properties. In practice, atomicity and serializability have been used primarily at the shared memory read/write level. Whereas linearizability has been used primarily at the method API level. Furthermore, existing tools for detecting serializability violations and atomicity violations do not check for quantitative properties.

VIII. CONCLUSIONS

We have presented a new algorithm for runtime checking of standard and quasi linearizability in concurrent data structures. Our method works directly on the C/C++ code and is fully automated, without requiring the user to write specifications or annotate linearization points. It guarantees that all the reported violations are real violations. We have implemented the new algorithm in a software tool called *Round-Up*. Our experimental evaluation shows that *Round-Up* is effective in detecting quasi linearizability violations and generating information for error diagnosis.

ACKNOWLEDGMENT

The authors would like to thank Christoph Kirsch and Michael Lippautz of University of Salzburg for making the *Scal* benchmarks available and for promptly answering our questions. Our work is supported in part by the NSF grant CCF-1149454 and the ONR grant N00014-13-1-0527.

REFERENCES

[1] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

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

[3] Y. Afek, G. Korland, and E. Yanovsky, "Quasi-Linearizability: Relaxed consistency for improved concurrency," in *International Conference on Principles of Distributed Systems*, 2010, pp. 395–410.

[4] T. A. Henzinger, A. Sezgin, C. M. Kirsch, H. Payer, and A. Sokolova, "Quantitative relaxation of concurrent data structures," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2013.

[5] A. Haas, M. Lippautz, T. A. Henzinger, H. Payer, A. Sokolova, C. M. Kirsch, and A. Sezgin, "Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation," in *Conf. Computing Frontiers*, 2013, p. 17.

[6] C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Performance, scalability, and semantics of concurrent fifo queues," in *International Conference on Algorithms and Architectures for Parallel Processing*, 2012, pp. 273–287.

[7] C. M. Kirsch and H. Payer, "Incorrect systems: it's not the problem, it's the solution," in *Proceedings of the Design Automation Conference*, 2012, pp. 913–917.

[8] H. Payer, H. Röck, C. M. Kirsch, and A. Sokolova, "Scalability versus semantics of concurrent fifo queues," in *ACM Symposium on Principles of Distributed Computing*, 2011, pp. 331–332.

[9] V. Vafeiadis, "Shape-value abstraction for verifying linearizability," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 335–348.

[10] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro, "Proving correctness of highly-concurrent linearisable objects," in *ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 129–136.

[11] Y. Liu, W. Chen, Y. A. Liu, and J. Sun, "Model checking linearizability via refinement," in *Proceedings of the 2nd World Congress on Formal Methods*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 321–337.

[12] M. T. Vechev, E. Yahav, and G. Yorsh, "Experience with model checking linearizability," in *International SPIN Workshop on Model Checking Software*, 2009, pp. 261–278.

[13] P. Cerný, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur, "Model checking of linearizability of concurrent list implementations," in *International Conference on Computer Aided Verification*, 2010, pp. 465–479.

[14] Y. Yang, X. Chen, and G. Gopalakrishnan, "Inspect: A runtime model checker for multithreaded C programs," University of Utah, Tech. Rep. UUCS-08-004, 2008.

[15] U. Salzburg, "Scal: High-performance multicore-scalable data structures. URL: http://scal.cs.uni-salzburg.at/."

[16] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-up: a complete and automatic linearizability checker," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 330–340.

[17] K. Adhikari, J. Street, C. Wang, Y. Liu, and S. Zhang, "Verifying a quantitative relaxation of linearizability via refinement," in *International SPIN Symposium on Model Checking of Software*. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 24–42.

[18] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[19] J. Aspnes, M. Herlihy, and N. Shavit, "Counting networks," *J. ACM*, vol. 41, no. 5, pp. 1020–1048, 1994.

[20] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[21] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003, pp. 338–349.

[22] A. Farzan and P. Madhusudan, "Causal atomicity," in *International Conference on Computer Aided Verification*, 2006, pp. 315–328.

[23] A. Sinha, S. Malik, C. Wang, and A. Gupta, "Predictive analysis for detecting serializability violations through trace segmentation," in *Formal Methods and Models for Codesign*, 2011, pp. 99–108.

[24] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," in *Parallel and Distributed Processing Symposium*, 2004.

[25] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 1–14.

[26] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: detecting atomicity violations via access interleaving invariants," in *Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 37–48.

[27] L. Wang and S. D. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Trans. Software Eng.*, vol. 32, no. 2, pp. 93–110, 2006.

[28] F. Chen and G. Rosu, "Parametric and sliced causality," in *International Conference on Computer Aided Verification*. Springer, 2007, pp. 240–253.

[29] A. Farzan and P. Madhusudan, "Monitoring atomicity in concurrent programs," in *International Conference on Computer Aided Verification*, 2008, pp. 52–65.

[30] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 293–303.

[31] C. Wang, S. Kundu, M. Ganai, and A. Gupta, "Symbolic predictive analysis for concurrent programs," in *International Symposium on Formal Methods*, 2009, pp. 256–272.

[32] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang, "Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis," in *SPIN workshop on Software Model Checking*, 2009.

[33] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2010, pp. 328–342.

[34] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *International Conference on Software Engineering*, 2011, pp. 221–230.

[35] A. Sinha and S. Malik, "Using concurrency to check concurrency: Checking serializability in software transactional memory," in *Parallel and Distributed Processing Symposium*, 2010.

[36] N. Sinha and C. Wang, "On interference abstractions," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2011, pp. 423–434.

[37] M. Said, C. Wang, Z. Yang, and K. Sakallah, "Generating data race witnesses by an SMT-based analysis," in *NASA Formal Methods*, 2011, pp. 313–327.

[38] C. Wang and M. Ganai, "Predicting concurrency failures in generalized traces of x86 executables," in *International Conference on Runtime Verification*, Sep. 2011.

[39] V. Kahlon and C. Wang, "Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs," in *International Conference on Computer Aided Verification*, 2010, pp. 434–449.

[40] J. Huang and C. Zhang, "Persuasive prediction of concurrency access anomalies," in *International Symposium on Software Testing and Analysis*, 2011, pp. 144–154.

[41] V. Kahlon and C. Wang, "Lock removal for concurrent trace programs," in *International Conference on Computer Aided Verification*, 2012, pp. 227–242.

[42] T.-F. Serbanuta, F. Chen, and G. Rosu, "Maximal causal models for sequentially consistent systems," in *International Conference on Runtime Verification*, 2012, pp. 136–150.

[43] J. Huang, J. Zhou, and C. Zhang, "Scaling predictive analysis of concurrent programs by removing trace redundancy," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, p. 8, 2013.