Round-Up: Runtime Verification of Quasi Linearizability for Concurrent Data Structures

Lu Zhang, Arijit Chattopadhyay, and Chao Wang, *Member, IEEE*

***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 introducing nondeterminism into the parallel computations quantitatively and then exploiting such nondeterminism to improve the runtime performance. However, ensuring the quantitative aspects of this correctness condition in the low-level code of the concurrent data structure implementation is a difficult task. Our runtime verification method is the first fully automated method for checking quasi linearizability in the C/C++ code of concurrent data structures. It guarantees that all the reported quasi linearizability violations manifested by the concurrent executions are real violations. We have implemented our method in a software tool based on the LLVM compiler and a systematic concurrency testing tool called Inspect. Our experimental evaluation shows that the new method is effective in detecting quasi linearizability violations in the source code implementations of concurrent data structures.**

***Index Terms*—Runtime verification, linearizability, serializability, atomicity, relaxed consistency, systematic concurrency testing, partial order reduction**

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 many 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 the often astronomically many thread interleavings. In practice, even a few hundred lines of highly concurrent 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 of these methods. This is extremely useful as a correctness condition for application developers because, as long as the program is correct while running in the sequential mode 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 almost always indicates that the implementation is buggy.

Quasi linearizability [3] is a quantitative relaxation of linearizability that has attracted a lot of attention in recent years [4], [5], [6], [7], [8]. For many highly parallel applications, the standard notion of linearizability often imposes unnecessary restrictions on the implementation, thereby leading to severe performance bottlenecks. Quasi linearizability has the advantage of preserving 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. That is, one can use a relaxed queue that allows some tasks to be overtaken occasionally if such relaxation leads to superior runtime 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, a unique identifier (id) generator does not need to be a perfect counter because 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 objects to have such occasional deviations from the standard semantics in exchange of higher performance.

While quasi linearizable concurrent data structures can have tremendous runtime performance advantages, ensuring the quantitative aspects of this correctness condition in the actual implementation is not an easy task. In this paper, we propose the first fully automated runtime verification method, called *Round-Up*, for checking *quasi* linearizability violations of concurrent data structures. To the best of our knowledge, prior to *Round-Up*, there does not exist any method for checking, for example, the `deq` operation of a relaxed queue is not overtaken by other `deq` operations for more than k times. Most of the existing concurrency bug detection methods focus on detecting simple bug patterns such as deadlocks, data-races, and atomicity violations, as opposed to quantitative properties manifested in *quasi* linearizability.

Our work also differs from the large body of work on checking standard linearizability, which is not a quantitative property. Broadly speaking, existing methods for checking *standard* linearizability fall into three groups. The first group consists of methods based on constructing mechanical proofs [9], [10], which typically require significant user intervention. The second group consists of automated methods based on techniques such as model checking [11], [12], [13], which work only on finite-state models or abstractions of the actual concurrent data structures. The third group consists of runtime verification tools that can directly check the im-

L. Zhang, A. Chattopadhyay, and C. Wang are with the Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University (Virginia Tech), Blacksburg, VA 24061, USA e-mail: chaowang@vt.edu.

plementation at the source-code level, but only for standard linearizability.

In contrast, *Round-Up* is the first runtime verification method for checking quasi linearizability in the source code implementations of concurrent data structures. It is fully automated in that the method does not require the user to provide functional specifications or to annotate the linearization points in the code. 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 also guarantees to report only real linearizability violations.

We have implemented the method in a software tool based on the LLVM compiler [14] and a systematic concurrency testing tool called Inspect [15], [16]. Our method can handle C/C++ programs that are written using the POSIX threads and GNU built-in atomic functions. Our experimental evaluation of the tool on a large set of concurrent data structure implementations shows that the new method is effective in detecting both standard and quasi linearizability violations. For example, we have found several real implementation bugs in the *Scal* suite [17], which is an open-source package that implements some recently published concurrent data structures. The bugs that we found in the *Scal* benchmarks have been confirmed by the developers.

To sum up, this paper makes the following contributions:

- We propose the first method for runtime checking of quasi linearizability in the source-code implementation of low-level concurrent data structures. The new method is fully automated and can guarantee that all the reported quasi linearizability violations are real violations.
- We have implemented the new method in a software tool based on LLVM and Inspect and evaluated the tool on the real C/C++ code of a large set of concurrent data structures. The results demonstrate that the new method is effective in detecting standard and quasi linearizability violations.

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. Figure 1 shows a multithreaded program that invokes the `enq` and `deq` methods of a queue, where `enq(i)` adds data item `i` to the end of the queue and `deq` removes the data item from the head. If Thread 2 executes `enq(3)` atomically, i.e., without interference from Thread 3, there will be three interleaved executions of the methods, all of which behave like a single-threaded execution. The sequential histories, shown in Figure 2, satisfy the standard FIFO semantics of the queue. Therefore, we call them *legal sequential histories*.

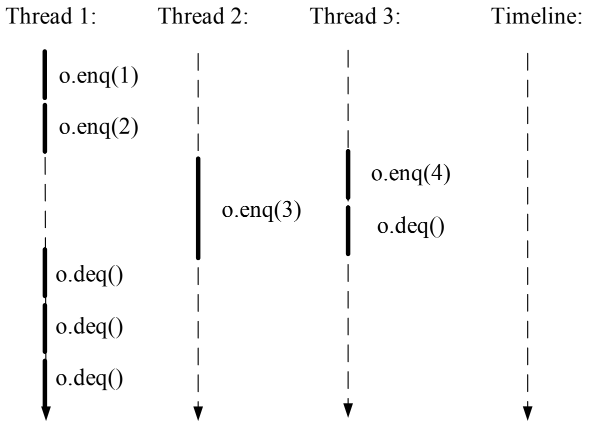


Fig. 1. A 3-threaded program that uses object o. Thread 1 starts by adding values 1 and 2 to the queue before creating two child threads. Then it waits for the child threads to terminate before removing another three data items. Here `enq(3)` runs concurrently with `enq(4)` and `deq()` in Thread 3.

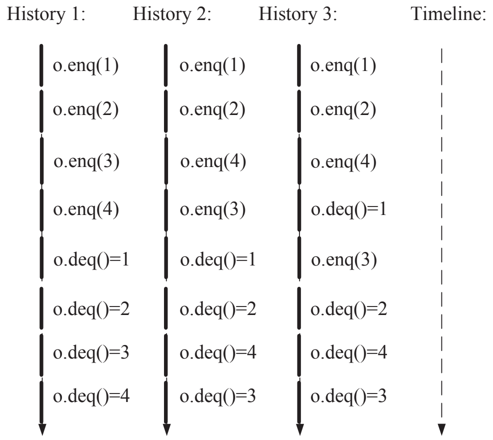


Fig. 2. The set of *legal sequential histories* generated by the program in Figure 1. These legal sequential histories form the *sequential specification*.

If the time interval of `enq(3)`, which starts at the method's 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 Figure 2. On the other hand, if the sequence of `deq` values is 3,2,1,4 in a concurrent history, for example, we say that the concurrent history has a linearizability violation, because the object no longer behaves like a FIFO queue.

However, being linearizable often means that the implementation has significant performance overhead, for example, 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 FIFO semantics can help

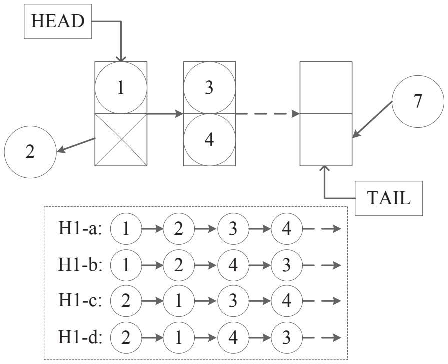


Fig. 3. An example implementation of *1-quasi linearizable* queue, where each of the linked list item is a segment that holds two data items. The first `deq` randomly returns a value from the set $\{1, 2\}$ and the second `deq` returns the remaining one. Then the third `deq` randomly returns a value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

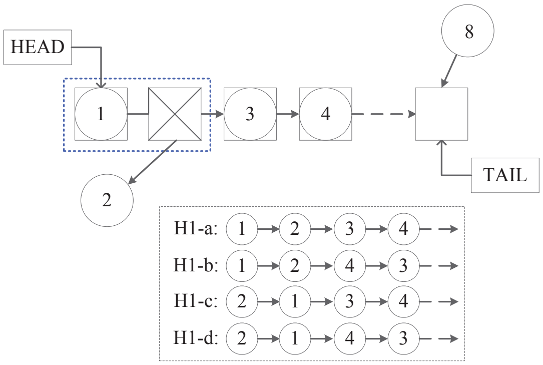


Fig. 4. An alternative implementation of *1-quasi linearizable* queue, which is based on the random-dequeued queue. The first `deq` randomly returns a value from $\{1, 2\}$ and the second `deq` returns the remaining one. Then the third `deq` randomly returns a value from the new window $\{3, 4\}$ and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

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 (Figure 3). In this case, 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 Figure 3. Among them, $H1$-a is linearizable. The other three are not linearizable, but they 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 such quasi linearizable concurrent 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 Figure 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, although it appears to be correct, the implementation would not be considered as 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 above example demonstrates the need for a new verification method that can help detect violations of the quantitative properties in *quasi* linearizable concurrent data structures. Unfortunately, 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 runtime verification method for detecting quasi linearizability violations in the source code of concurrent data structures.

III. Preliminaries

A. Linearizability

We follow the notation in [1], [2] to define a *history* as a sequence of events, denoted $h = e_1 e_2 \ldots e_n$, where each event is either a method invocation or a method response for an object. When there are multiple objects involved in the history h, we use $\rho = h|o$ to denote the projection of h to the object o, which results in a subsequence of events related only to this 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 this 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 $i = 1, \ldots, k$. Therefore, 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 the history ρ. Let $\rho[e]$ be the index of the event e in ρ. For two events e_1 and e_2, we say that $e_1 <_\rho e_2$ if and only if $\rho[e_1] < \rho[e_2]$.

Definition 1: A *sequentialization* 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 $(e_i <_{\rho'} e_j)$. That is, the non-overlapping method calls in ρ retain their execution order in ρ', whereas the overlapping method calls in ρ may take effect in any order in ρ'.

A *sequential specification* of object o, denoted $spec(o)$, is the set of all *legal* sequential histories – they are histories that conform to the semantics of the object. For example, a legal sequential history of a queue is one in which 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 sequentialization ρ' such that $\rho' \in spec(o)$. In other words, as long as the concurrent history ρ can be mapped to at least one legal sequential history $\rho' \in spec(o)$, it is considered as linearizable.

B. Quasi Linearizability

The notion of quasi linearizability relies on the permutation distance between two sequential histories. Let $\rho' = e'_1 e'_2 \ldots e'_n$ be a permutation of $\rho = e_1 e_2 \ldots e_n$. 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 $Domain(o)$. Let $D \subset Powerset(Domain(o))$ be a subset of the powerset.

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 always follow the FIFO order, but `deq` values may be out-of-order by at most K steps, can be specified as follows:

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

Here, $\langle \texttt{o.enq}(x), void\rangle$ represents the invocation of $\texttt{o.enq}(x)$ and the return value $void$, where $x \in X$ is the data value added to the queue. Similarly, $\langle \texttt{o.deq}(), x\rangle$ represents the invocation of $\texttt{o.deq}()$ and the return value x. $Q_{\texttt{queue}}(D_{\texttt{enq}}) = 0$ means that the `enq` events follow the standard FIFO order. $Q_{\texttt{queue}}(D_{\texttt{deq}}) = K$ means that the `deq` events are allowed to be out-of-order by at most K steps. Such relaxed queue can be useful in producer-consumer applications, for example, when data are enqueued by a single producer and dequeued by multiple consumers. By relaxing the ordering of the dequeue operations alone, we allow at most K consumers to retrieve data from the queue simultaneously, without any memory contention. The relaxed semantics allows more freedom in the implementation of the concurrent queue, thereby leading to significant performance improvement [4], [8].

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 sequentialization 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 the sequentialization ρ' such that
  - $\rho'' \in spec(o)$; and
  - $\Delta(\ \rho'|d,\ \rho''|d\ ) \le Q_o(d)$ for all subset $d \in D$.

In other words, ρ' needs to be a legal sequential history by itself or be within a bounded distance from a legal sequential history ρ''.

From now on, given a sequential history ρ', we call $\Psi = \{\ \rho'' \mid \Delta(\ \rho'|d,\ \rho''|d\ ) \le Q_o(d)$ 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. Furthermore, the standard notion of linearizability is subsumed by quasi linearizability with $Q_o : D \to 0$.

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:

  $\exists$ sequentialization 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:

  $\forall$ test 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 test program P that uses the concurrent object o, 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 quasi linearizable.

IV. Overall Algorithm

The overall algorithm for checking quasi linearizability consists of two phases (see Figure 5). In Phase 1, we systematically execute the test program P together with a standard data structure implementation 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 implementation, and for each concurrent history ρ, check whether ρ is quasi linearizable.

For widely used 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 implementation to 0, which effectively turns it into a normal queue.

In Phase 1, we use a systematic concurrency testing tool called Inspect [15], [16] to compute all the legal sequential histories. Given a multithreaded C/C++ program and fixed data input, Inspect can be used to systematically generate all possible thread interleavings of the program under the data input. In order to use Inspect in Phase 1, we have modified Inspect to automatically wrap up every method call of a shared object o 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. We leverage these *legal sequential histories* to construct the sequential specification $spec(o)$ of the object o. For any given test program P, the sequential specification $spec(o)$ is represented as the set of all legal sequential histories of the test program P. We focus on checking the linearizability of each individual concurrent object without loss of generality, because linearizability is compositional in that an execution history is linearizable with respect to multiple objects if it is linearizable with respect to each individual object.

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 thread functions such as mutex lock/unlock and signal/wait, we have extended Inspect to support the set of GNU built-in functions for atomic memory access, which are frequently used in practice for implementing concurrent data structures. Since we use LLVM as the front-end for code instrumentation, it means that we treat LLVM atomic instructions such as `cmpxchg` and `atomicrmw` similar to shared variable write instructions. Here, `cmpxchg` refers to the atomic compare-and-exchange instruction, and `atomicrmw` refers to the atomic read-modify-write instruction. During Phase 2, Inspect will interleave them systematically while generating the concurrent histories.

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

The pseudocode 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). For ease of presentation, we assume that $O_o(d) = K$ for all subsets $d \in Powerset(Domain(o))$, where K is an integer constant. The main challenge in Algorithm 1 is to generate the set Φ of sequentializations of the given history ρ and the set Ψ of quasi permutations of each $\rho' \in \Phi$. The first step will be explained in the remainder of this section. The second step, which is significantly more involved, will be explained in the next section.

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

```
check_quasi_linearizability ( $\rho, K$ )
{
    $\Phi \leftarrow$ compute_sequentializations( $\rho$ );
    for each ( $\rho' \in \Phi$ ) {
        if ( $\rho' \in spec(o)$ ) return TRUE;
        $\Psi \leftarrow$ compute_quasi_permutations( $\rho', K$ );
        for each ( $\rho'' \in \Psi$ ) {
            if ( $\rho'' \in spec(o)$ ) return TRUE;
        }
    }
    return FALSE;
}
compute_sequentializations ( $\rho$ )
{
    $\Phi \leftarrow \{\rho\}$;
    while ( $\exists$ a concurrent history $\rho_0 \in \Phi$ ) {
        Let $\rho_0 = \varphi\ \mathtt{inv}_1\ \mathtt{inv}_2\ \phi\ \mathtt{resp}_1\ \psi\ \mathtt{resp}_2\ \ldots$;
        $\rho_1 \leftarrow \varphi\ \mathtt{inv}_1\ \mathtt{resp}_1\ \mathtt{inv}_2\ \phi\ \psi\ \mathtt{resp}_2\ \ldots$;
        $\rho_2 \leftarrow \varphi\ \mathtt{inv}_2\ \mathtt{resp}_2\ \mathtt{inv}_1\ \phi\ \mathtt{resp}_1\ \psi\ \ldots$;
        $\Phi \leftarrow \Phi \cup \{\rho_1, \rho_2\} \setminus \{\rho_0\}$;
    }
    return $\Phi$;
}
compute_quasi_permutations ( $\rho', K$ )
{
    $\Psi \leftarrow \{\ \}$;
    state_stack $\leftarrow$ first_run( $\rho', K$ );
    while ( TRUE ) {
        $\rho'' \leftarrow$ backtrack_run ( state_stack, $\rho'$ );
        if ( $\rho''$ = = null ) break;
        $\Psi \leftarrow \Psi \cup \{\rho''\}$;
    }
    return $\Psi$;
}
```

We now explain the detailed algorithm for computing the set Φ of sequentializations for the given history ρ. The computation is carried out by Subroutine `compute_sequentializations`(ρ). Let a history $\rho_0 = \varphi\ \mathtt{inv}_1\ \mathtt{inv}_2\ \phi\ \mathtt{resp}_1\ \psi\ \mathtt{resp}_2\ \ldots$ where φ, ϕ and ψ are arbitrary subsequences and $\mathtt{inv}_1, \mathtt{inv}_2$ 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 $(\mathtt{inv}_i, \mathtt{resp}_i)$ and $(\mathtt{inv}_j, \mathtt{resp}_j)$ in ρ, if they do not overlap in time, meaning that either $\mathtt{resp}_i <_\rho \mathtt{inv}_j$ or $\mathtt{resp}_j <_\rho \mathtt{inv}_i$, we will keep this execution order. But if they overlap in time, we will generate two new histories, where one has $\mathtt{resp}_i <_\rho \mathtt{inv}_j$ and the other has $\mathtt{resp}_j <_\rho \mathtt{inv}_i$.

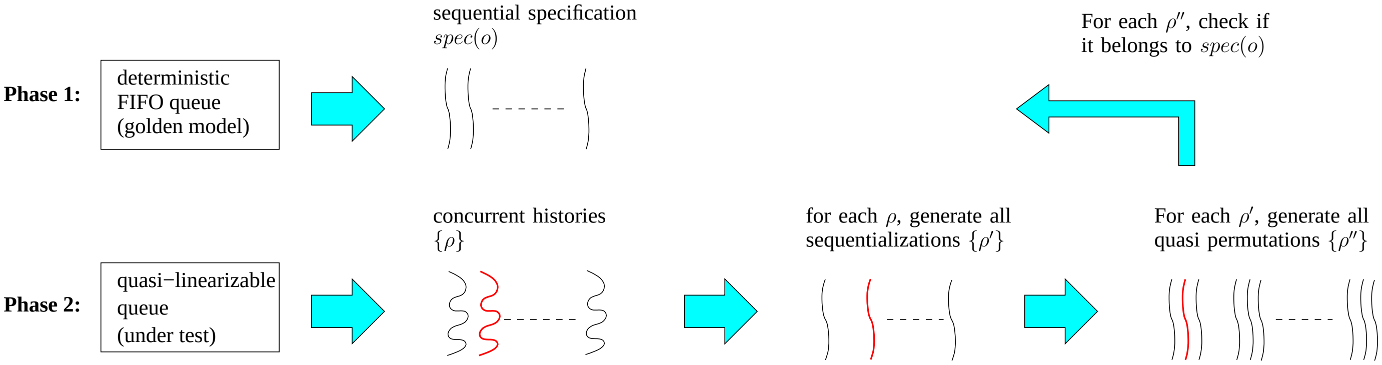

Phase 1:
deterministic FIFO queue (golden model)
sequential specification
$spec(o)$
For each ρ'', check if it belongs to $spec(o)$
Phase 2:
quasi−linearizable queue (under test)
concurrent histories
$\{\rho\}$
for each ρ, generate all sequentializations $\{\rho'\}$
For each ρ', generate all quasi permutations $\{\rho''\}$

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

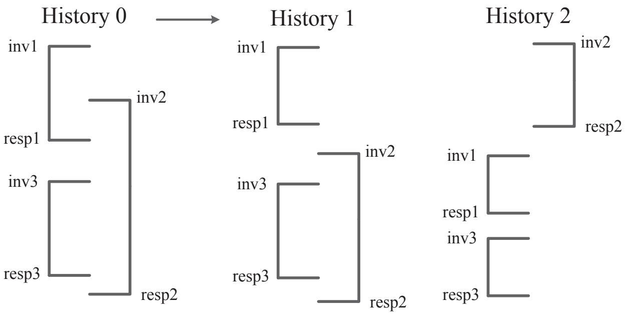


Fig. 6. Example: Computing *sequentializations* of a given concurrent history by repeatedly sequentializing the first two overlapping method calls denoted by `(inv1,resp1)` and `(inv2,resp2)`.

Example. Consider the history in Figure 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 $\texttt{resp}_2$. 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`.

The complexity of `compute_sequentializations`(ρ) depends on the length of the input history ρ, as well as the number of overlapping method calls. Let M denote the length of the history ρ and L denote the number of overlapping method calls (where $L \leq M$). The complexity for computing all sequentializations of ρ is $O(M \times 2^L)$ in the worst case. In practice, however, this subroutine will not become a performance bottleneck for two reasons. First, to expose linearizability violations, small test programs with few concurrent method calls often suffice, which means that M is small. Second, the input to our method, $\rho = h|o$, is a subsequence of the original history projected to the object o, thereby consisting of only the events related to object o. Since linearizability is inherently compositional, we can check it for each individual object in isolation.

After computing the set Φ of sequentializations, we check if any $\rho' \in \Phi$ is a legal sequential history, as shown at Line 4 in Algorithm 1. According to Definition 1, as long as one sequentialization $\rho' \in \Phi$ 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 a sequentialization $\rho' \in \Phi$ is 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`: $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \dots s_n \xrightarrow{e_n}$. Each state s_i, where $i = 1, \dots, n$, 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 all possible permutations of ρ', denoted $\{\rho''\}$, such that the distance between ρ' and ρ'' is bounded by the quasi factor K. Our method for constructing the quasi permutations follows the *strict out-of-order* semantics as defined in [3], [7]. 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.

To illustrate the *strict out-of-order* definition, consider the 1-quasi queue below, where the sequentialized history ρ' is `deq()=1,deq()=2,deq()=3`.

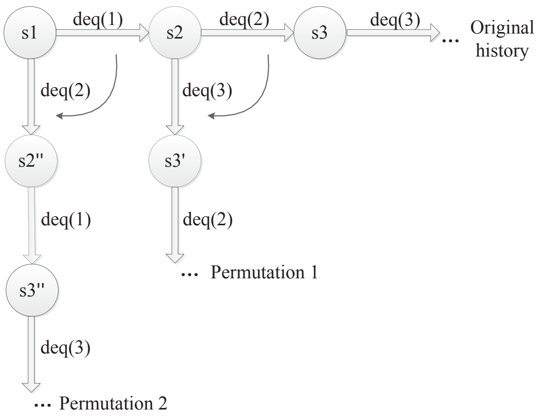


Fig. 7. An example search tree for generating all 1-quasi permutations of the input sequence `deq(1);deq(2);deq(3)`.

```
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
```

The input history can be arbitrarily re-shuffled to produce five additional histories, of which only History 1 and History 2 satisfy the above two restrictions of the *strict out-of-order* 1-quasi permutation. 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 Figure 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 initial run, we construct the state stack that holds the initial history. Then we find the last backtrack state, which is state s_2, where we can 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.

An important feature of our algorithm is that it directly constructs valid permutations while skipping the invalid ones, as opposed to constructing all permutations and then filtering out the invalid ones. In the running example, for instance, our method will directly generate History 1 and History 2 from History 0, while skipping 3, 4, and 5.

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)}	≥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$\setminus s$.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$\setminus s_2''$.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 $\backslash$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.

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

```
first_run ( $\rho', K$ ) {
    state_stack $\leftarrow$ empty list;
    for each ( event $ev$ in the sequence $\rho'$ ) {
        $s \leftarrow$ new state;
        state_stack.append( $s$ );
        $s$.select $\leftarrow ev$;
        $s$.done $\leftarrow \{ev\}$;
        init_enabled_and_lateness( $s, ev, K$ );
    }
    return state_stack;
}
init_enabled_and_lateness ( $s, ev, K$ ) {
    lateness $\leftarrow$ 0;
    while( 1 ) {
        $s$.enabled.add( $\langle ev$,lateness$\rangle$ );
        if( lateness == $-k$ || $s$.prev == null ) {
            $s$.newly_enabled.add( $\langle ev$, lateness$\rangle$ );
            break;
        }
        lateness$--$;
        $s \leftarrow s$.prev in state_stack;
    }
}
backtrack_run ( state_stack, $\rho$ ) {
    Let $s$ be the last state in state_stack such that
        pick_an_enabled_event( $s$ ) $\neq$ null;
    if( such $s$ does not exist )
        return null;
    for each( state after $s$ in state_stack ) {
        reset $s$.select, $s$.done, and $s$.enabled,
        but keep $s$.newly_enabled;
    }
    while( $s \neq$ null ) {
        $ev \leftarrow$pick_an_enabled_event ( $s$ );
        $s$.select $\leftarrow ev$;
        $s$.done $\leftarrow \{ev\}$;
        $s \leftarrow s$.next;
        update_enabled_and_lateness( $s$ );
    }
    return (sequence of selected events in state_stack);
}
pick_an_enabled_event ( $s$ ) {
    if( $\exists\langle ev, lateness\rangle \in s$.enabled && $lateness = k$) {
        if( $ev \notin s$.done )      // must-select event
            return $ev$;
        else
            return null;
    }
    if( $\exists\langle ev, lateness\rangle \in s$.enabled && $ev \notin s$.done) )
        return $ev$;
    else
        return null;
}
update_enabled_and_lateness ( $s$ ) {
    $p \leftarrow s$.prev;
    if( $s$ or $p$ do not exist )
        return;
    $s$.enabled $\leftarrow$ { };
    for each( $\langle ev, lateness\rangle \in p$.enabled && $ev \notin p$.done) {
        $s$.enabled.add( $\langle ev, lateness--\rangle$ );
    }
    for each( $\langle ev, lateness\rangle \in s$.newly_enabled ) {
        $s$.enabled.add( $\langle ev, lateness\rangle$ );
    }
}
```

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'.

The complexity of `compute_quasi_permutations()` depends on the length of the input history ρ', denoted M, as well as the quasi factor K. The overall complexity is $O((1+K)!^{M/(1+K)})$, where $(1+K)!$ is the number of permutations of $(1+K)$ events, and $M/(1+K)$ is the number of groups of size $(1+K)$ in a history of M events. In practice, this subroutine will not become a performance bottleneck because test programs with small M often suffice in revealing linearizability violations. In addition, the input $\rho' = h|o$ is a project of the original history h to each individual object o, which further reduces the size of the input.

D. Discussions

The real performance bottleneck in practice is due to the potentially large number of concurrent histories that need to be fed to our method as input, as opposed to the computational overhead of Phases 1 and 2 within our method. The reason why we may have a large number of concurrent histories is because the number is exponential in the number of *low-level concurrent operations* in the test program. Typically, the number of these low-level operations, denoted N, can be significantly larger than M, the number of method calls in the same history, because each method has to be implemented by many low-level operations such as Load/Store over shared memory and thread synchronizations.

In our experiments, we try to avoid the performance bottleneck by reducing the number of concurrent histories, primarily through the use of small test programs as input, and advanced exploration heuristics (such as context bounding) in the Inspect tool.

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. Furthermore, at this moment, the erroneous execution trace that gives rise to ρ has been logged into a disk file by the underlying Inspect tool. The execution trace contains all the method invocation and response events in ρ, as well as the low-level concurrent operations, such as Load/Store operations and thread synchronizations. Such trace can be provided to the user as debugging aid. As an example, we will show in Section VI-D how it can help the user diagnose a real bug found in the *Scal* benchmarks.

However, since our method implements the L1 and L2 checks but not the L3 check as defined in Section III, 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 legal sequential 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 given to our method 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 *Inspect* for systematically generating interleaved executions. Our tool, called *Round-Up*, can handle C/C++ code of concurrent data structures on the Linux/PThreads platform. We have extended the original implementation of *Inspect* [15], [16] by adding the support for GNU built-in atomic functions for direct access of shared memory, since in practice, they are frequently used in the low-level code for implementing concurrent data structures.

We have conducted experiments on a set of concurrent data structures [3], [5], [4], [8], [5], [7] including both standard and quasi linearizable queues, stacks, and priority queues. For some of these concurrent data structures, there are several variants, each of which uses a different implementation scheme. These benchmark examples fall into two sets.

A. Results on the First Set of Benchmarks

The characteristics of the first set of 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 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. For example, 3*4+1 in the *calls* column means that one child thread issued 3 method calls, the other child thread issued 4 method calls, and the main thread issued 1 method call. 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 sequentializations, and the time for checking them.

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)	sequentialization	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

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 set the quasi factor to 2 unless specified otherwise in the test program name (e.g., qfactor=3).

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.* [18] 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.

B. Experimental Results on the Scal *Benchmarks*

We have also conducted experiments on a set of recently released high-performance concurrent objects in the *Scal* suite [17]. The characteristics of this second set of benchmark programs are shown in Table IV. The format of this table is the same as Table I. For a more detailed description of the individual algorithms, please refer to the original papers where these concurrent data structures are published [7], [5], [6].

Table V shows our experimental results for applying *Round-Up* to this set of benchmarks. In addition to obtaining the performance data, 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. Note that the fact that *sl-queue* is not linearizable is known.

In contrast, *k-stack* is designed to be quasi linearizable, but due to an ABA bug, the data structure implementation is not quasi linearizable. In fact, it is not even linearizable since the behavior may violate the standard functional specification of a stack. (An ABA problem [2] occurs in a multithreaded program when a memory location is read twice, has the same values for both reads, and therefore appears to indicate “nothing has changed.” However, another thread may have interleaved in between the two reads, changed the value, did

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 [17] 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

some work, and then changed the value back.)

Our tool is able to quickly detect the linearizability violation in *sl-queue* and the quasi linearizability violation in *k-stack*. We have reported the violation 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 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.

C. Optimizations to Reduce the Runtime Overhead

To further improve the runtime performance, we have applied the following optimizations in Phases 1 and 2, to reduce the number of traces generated from the sequential specification $spec(o)$, the set Φ of sequentializations, and the set Ψ of quasi permutations.

Specifically, our optimizations are as follows: (1) in Phase 1, we remove from $spec(o)$ all identical serial histories generated by the Inspect tool by simply comparing their textual representations; (2) in Phase 2, we remove from the input all identical concurrent executions fed to our method—we say that two concurrent executions are identical if they give rise to the same history, even though they may differ in the low-level concurrent operations within the method calls; and (3) for each sequentialization $\rho' \in \Phi(\rho)$, or each quasi permutation $\rho'' \in \Psi(\rho')$, we check if ρ' or ρ'' is identical to some sequentialization or quasi permutation of a previously explored concurrent history, and if the answer is yes, we skip it.

It is worth pointing out that, in our application, the length of the sequential and concurrent execution histories are often short. Therefore, a straightforward implementation of the above algorithm for identifying redundant histories already works well in practice. For example, in Table V, each execution history has at most 14 method calls (1+1+12), among which only two method calls are executed concurrently. In such case, the pair-wise comparison of execution histories as described above is fast. Specifically, compared to the cost of the other parts of our method, e.g., loading the executable to the main memory and performing the interleaved executions, the time spent on comparing these histories in memory is always negligible. This is the reason why, in Table V, we only report the number of permutations and the total time.

Our evaluation of these optimizations on the *Scal* benchmarks shows that they can lead to a significant reduction in the number of permutations and the execution time. Our

TABLE V
RESULTS OF CHECKING QUASI LINEARIZABILITY FOR THE SCAL [17] 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+12	NO	2	0.38	0/2	1032	1.61
sl-queue (enq+enq)	2/3	1*1+12	YES	2	0.37	1/2	1032	1.78
sl-queue (deq+deq)	2/3	1*1+12	YES	2	0.37	4/8	5160	7.21
t-stack (push+pop)	2/3	1*1+12	NO	2	0.58	0/8	5160	7.77
t-stack (push+push)	2/3	1*1+12	NO	2	0.59	0/8	5160	7.78
t-stack (pop+pop)	2/3	1*1+12	NO	2	0.56	0/8	5160	7.63
ms-queue (enq+deq)	2/3	1*1+12	NO	2	0.76	0/3	1720	2.94
ms-queue (enq+enq)	2/3	1*1+12	NO	2	0.79	0/31	20984	32.47
ms-queue (deq+deq)	2/3	1*1+12	NO	2	0.76	0/12	7912	12.33
rd-queue (enq+deq)	2/3	1*1+12	NO	2	0.90	0/5	3096	5.97
rd-queue (enq+enq)	2/3	1*1+12	NO	2	0.91	0/31	20984	37.83
rd-queue (deq+deq)	2/3	1*1+12	NO	2	0.87	0/4	2408	7.31
bk-queue (enq+deq)	2/3	1*1+12	NO	2	1.27	0/29	19608	34.45
bk-queue (enq+enq)	2/3	1*1+12	NO	2	1.30	0/41	27864	50.98
bk-queue (deq+deq)	2/3	1*1+12	NO	2	1.22	0/24	16168	28.11
ubk-queue (enq+deq)	2/3	1*1+12	NO	2	2.42	0/21	14104	35.71
ubk-queue (enq+enq)	2/3	1*1+12	NO	2	1.86	0/41	27864	61.94
ubk-queue (deq+deq)	2/3	1*1+12	NO	2	1.84	0/52	35432	68.21
k-stack (push+pop)	2/3	1*1+12	YES	2	1.55	11/69	47128	1m29s
k-stack (push+push)	2/3	1*1+12	NO	2	1.69	0/12	7192	15.13
k-stack (pop+pop)	2/3	1*1+12	NO	2	1.62	0/8	5160	9.46

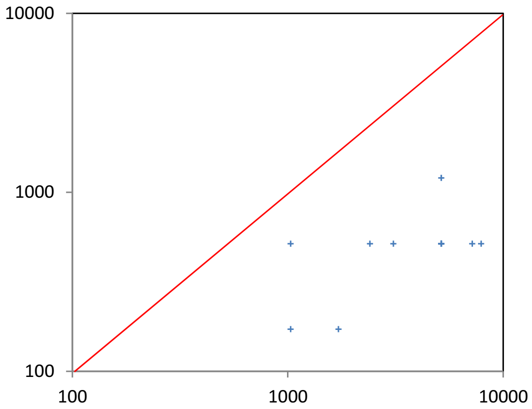


Fig. 8. Results: Evaluating the impact of optimizations on the performance for *Scal* benchmarks [17]. We compare the *number of quasi permutations* generated by our method with optimizations (y-axis) and without optimizations (x-axis). Each + represents a test case. Test cases below the diagonal line are the winning cases for our method with optimizations.

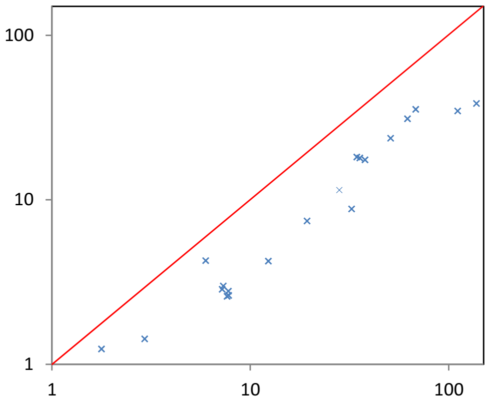


Fig. 9. Results: Evaluating the impact of optimizations on the performance for *Scal* benchmarks [17]. We compare the *execution time (in seconds)* of our method with optimizations (y-axis) and without optimizations (x-axis). Each + represents a test case. Test cases below the diagonal line are the winning cases for our method with optimizations.

experimental results are shown in the scatter plots in Figure 8 and 9, which compare the number of permutations and the total time of our method with and without the optimizations. Note that the concrete time and the number of permutations, with optimizations, have already been reported in the last two columns of Table V. The x-axis shows the number of permutations and the execution time without these optimizations, whereas the y-axis shows the number of permutations and the execution time with these optimizations. Here, points below the red diagonal lines are the winning cases for the optimizations. Note that both the x-axis and the y-axis are in logarithmic scale.

D. Generating Diagnosis Information

Our *Round-Up* tool guarantees that all the reported (quasi) linearizability violations are real violations; that is, they can appear in the actual program execution. To help the developer diagnose the reported violation, our tool leverages Inspect, the underlying concurrency testing tool, to generate a detailed execution trace for each concurrent history. When a current history ρ is proved to be erroneous, i.e., neither linearizable nor quasi linearizable, the detailed execution trace will be provided to the user as debugging aid. In this execution trace, we have recorded not only the method invocation and response events in ρ but also the low-level operations by each thread, including Load/Store instructions over shared memory and thread synchronizations. For more information on how such execution trace is generated, please refer to the Inspect tool [15], [16].

In terms of the ABA bug that we have detected in *k-stack*, for example, our tool shows that the violation occurs when one thread executes the `push` operation while another thread executes the `pop` operation concurrently. Due to erroneous thread interleaving, it is possible for the same data item to be added to the stack twice, despite the fact that the `push` operation is executed only once. In the remainder of this subsection, we shall explain this bug in more details.

First, we show in Algorithm 3 the simplified pseudo code of the `push` and `pop` methods of *k-stack* as implemented in the *Scal* suite. The `push()` method starts by putting the data item into the current segment and then trying to commit, by checking if the flag `segment→delete` is 0. The `pop()` method, which may be executed by another thread concurrently with `push()`, starts by setting the flag `segment→delete` to 1 and then trying to remove the segment if it is empty.

The bug happens when the top segment is empty, thread T_1 is trying to push a data item onto the stack, and at the same time, thread T_2 is trying to pop a data item from the stack. The erroneous execution trace is summarized as follows:

```
Thread 1                  Thread 2
--------                  --------
                          Pop:L10-12
Push:L3-L5
                          try_remove_segment:L23
Committed:L17
                          try_remove_segment:L24-L25
Committed:L18-L19
Push:L3-L6
--------                  --------
```

Algorithm 3 The simplified pseudo code of *k-stack* in *Scal*

```
1: push(T item)
2: {
3:     while ( TRUE ) {
4:         put item into the segment;
5:         if ( committed() ) return TRUE;
6:     }
7: }
8: pop()
9: {
10:    while ( TRUE ) {
11:        try to find an item in the segment;
12:        if (not found) try_remove_segment();
13:    }
14: }
15: committed()
16: {
17:    if ( segment→delete==0 ) return TRUE;
18:    else if ( segment→delete==1 ) {...}
19:    return FALSE;
20: }
21: try_remove_segment()
22: {
23:    segment→delete=1;
24:    if ( is_empty(segment) ) {...}
25:    else  segment→delete=0;
26: }
```

From the above debugging information reported by our *Round-Up* tool, we can explain the erroneous thread interleaving as follows:

- **Step 1**: Since the stack is empty, inside method `pop`, thread T_2 cannot find any data item in the segment, and therefore invokes function `try_remove_segment()` to try to remove the segment;
- **Step 2**: Inside method `push`, thread T_1 pushes a data item into the segment, and then calls function `committed()` to make sure the data item is successfully added by waiting `committed()` to return true;
- **Step 3**: Inside function `try_remove_segment()`, thread T_2 marks the top segment as *deleted*;
- **Step 4**: Inside function `committed()`, thread T_1 fails the condition in Line 17;
- **Step 5**: Inside function `try_remove_segment()`, thread T_2 checks the segment again, and notices that the segment is not empty. Therefore, thread T_2 sets the top segment back to *not deleted*;
- **Step 6**: Inside function `committed()`, thread T_1 fails the condition in Line 18 and returns false. Therefore, method `push` returns to the beginning of the while loop, and pushes the same data item into the segment again.

As we can see, this bug requires complex interactions between the two threads to manifest and therefore is difficult to detect manually. However, it can be easily detected by our *Round-Up* tool, since our tool has the capability of systematically enumerating possible thread interleavings and then checking each of them for (quasi) linearizability violations.

VII. Related Work

Quasi linearizability was first introduced by Afek *et al.* [3] as a way to trade off correctness for better runtime performance while implementing concurrent queues for highly parallel applications. Since then, the idea of quantitatively relaxing the consistency requirement of linearizability to improve runtime performance have been used in the design of various concurrent data structures [8], [5], [6], [4]. More recently, Henzinger *et al.* [7] generalized the idea by proposing a systematic and formal framework for obtaining new concurrent data structures by quantitatively relaxing existing ones.

Round-Up is the first runtime verification method for detecting quasi linearizability violations in the source code implementation of concurrent data structures. This paper is an extended version of our recent work [19] where we proposed the method for the first time. In this extended version, we have provided a more detailed description of the algorithms and presented more experimental results.

A closely related work is a model checking based method that we proposed [20], [21] for verifying quantitative relaxations of linearizability in *models* of concurrent systems. However, the method was designed for statically verifying finite-state models, not for dynamically checking the C/C++ code of concurrent data structure implementations. Although the Line-Up tool by Burckhardt *et al.* [18] also has the capability of dynamically checking the code of concurrent data structures, it can only check linearizability but not *quasi* linearizability. In contrast, the main contribution of this paper is proposing a new method for checking quasi linearizability.

Besides Line-Up [18], there is a large body of work on statically verifying standard linearizability properties. 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 are also techniques for verifiying linearizability by constructing mechanical proofs, often with manual intervention [9], [10], [22]. However, none of these methods can check quantitative relaxations of linearizability.

There are also static and runtime verification methods for other types of consistency conditions, including sequential consistency [23], quiescent consistency [24], and eventual consistency [25]. Some of these consistency conditions, in principle, may be used during software testing and verification to ensure the correctness of concurrent data structures. However, they are not as widely used as linearizability in this application domain. Furthermore, they do not have the same type of quantitative properties and the corresponding verification challenges as in *quasi* linearizability.

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 criteria. In the literature, there is a large body of work on detecting serializability and atomicity violations (e.g. [26], [27], [28] and [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48]). It is worth pointing out that these bug finding methods differ from our 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 instruction level, whereas linearizability has been used primarily at the method API level. Furthermore, existing tools for detecting serializability and atomicity violations do not check for quantitative properties.

VIII. Conclusions

We have presented a new algorithm for runtime verification 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 functional specifications or to annotate linearization points in the code. It also 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 in generating information for error diagnosis.

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[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] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke, "LLVM: A low-level virtual instruction set architecture," in *ACM/IEEE international symposium on Microarchitecture*, San Diego, California, Dec 2003.

[15] 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.

[16] Y. Yang, X. Chen, G. Gopalakrishnan, and R. Kirby, "Efficient stateful dynamic partial order reduction," in *SPIN Workshop on Model Checking Software*, 2008, pp. 288–305.

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

[18] 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.

[19] L. Zhang, A. Chattopadhyay, and C. Wang, "Round-Up: Runtime checking quasi linearizability of concurrent data structures," in *IEEE/ACM International Conference On Automated Software Engineering*, 2013, pp. 4–14.

[20] K. Adhikari, J. Street, C. Wang, Y. Liu, and S. Zhang, "Verifying a quantitative relaxation of linearizability via refinement," *International Journal on Software Tools for Technology Transfer*, 2015.

[21] ——, "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.

[22] H. Zhu, G. Petri, and S. Jagannathan, "Poling: SMT aided linearizability proofs," in *International Conference on Computer Aided Verification*, 2015, pp. 3–19.

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

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

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

[26] 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.

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

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

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

[42] 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.

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

[44] 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.

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

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

[47] 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.

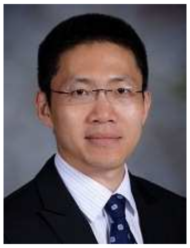
[48] J. Huang, J. Zhou, and C. Zhang, “Scaling predictive analysis of concurrent programs by removing trace redundancy,” *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, p. 8, 2013.

Lu Zhang received the B.S. degree from Tsinghua University, Beijing, China, in 2010. He is currently pursuing the Ph.D. degree from the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA.

His current research interests include software engineering and formal methods, with a focus on developing automated methods and tools for verification and debugging of critical software systems.

Arijit Chattopadhyay received the B.Tech. degree in Computer Science from National Institute of Technology, Durgapure, India, in 2009, and the M.S. degree from the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA, in 2014.

He is currently a software engineer at Bloomberg LP, New York, USA.



Chao Wang (M’02) received the Ph.D. degree from the University of Colorado, Boulder, CO, USA, in 2004.

He is currently an Assistant Professor with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA. He has published a book and over 50 papers in top venues in the areas of software engineering and formal methods.

Dr. Wang received the FMCAD Best Paper Award in 2013, the ACM SIGSOFT Distinguished Paper Award in 2010, and the ACM TODAES Best Paper of the Year Award in 2008. He was a recipient of the NSF Faculty CAREER Award in 2012 and the ONR Young Investigator Award in 2013.