

# Faster linearizability checking via *P*-compositionality<sup>\*</sup>

Alex Horn and Daniel Kroening

University of Oxford

**Abstract.** Linearizability is a well-established consistency and correctness criterion for concurrent data types. An important feature of linearizability is Herlihy and Wing’s locality principle, which says that a concurrent system is linearizable if and only if all of its constituent parts (so-called objects) are linearizable. This paper presents *P-compositionality*, which generalizes the idea behind the locality principle to operations on the same concurrent data type. We implement *P*-compositionality in a novel linearizability checker. Our experiments with over nine implementations of concurrent sets, including Intel’s TBB library, show that our linearizability checker is one order of magnitude faster and/or more space efficient than the state-of-the-art algorithm.

## 1 Introduction

*Linearizability* [1] is a well-established correctness criterion for concurrent data types and it corresponds to one of the three desirable properties of a distributed system, namely *consistency* [2]. The intuition behind linearizability is that every operation on a concurrent data type is guaranteed to take effect instantaneously at some point between its call and return.

The significance of linearizability for contemporary distributed key/value stores has been highlighted recently by the *Jepsen* project, an extensive case study into the correctness of distributed systems.<sup>1</sup> Interestingly, Jepsen found linearizability bugs in several distributed key/value stores despite the fact that they were designed based on formally verified distributed consensus protocols. This illustrates that there is often a gap between the design and the implementation of distributed systems. This gap motivates the study in this paper into runtime verification techniques (in the form of so-called *linearizability checkers*) for finding linearizability bugs in a single run of a concurrent system.

The input to a linearizability checker consists of a sequential specification of a data type and a certain partially ordered set of operations, called a *history*. A history represents a single terminating run of a concurrent system. We assume that the concurrent system is deadlock-free since there already exist good deadlock detection tools. Despite the restriction to single histories, the problem of

<sup>\*</sup> This work is funded by a gift from Intel Corporation for research on Effective Validation of Firmware and the ERC project ERC 280053.

<sup>1</sup> <https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul>

checking linearizability is NP-complete [3]. This high computational complexity means that writing an efficient linearizability checker is inherently difficult. The problem is to find ways of pruning a huge search space: in the worst case, its size is  $O(N!)$  where  $N$  is the length of the run of a concurrent system.

This paper presents a novel linearizability checker that efficiently prunes the search space by partitioning it into independent, faster to solve, subproblems. To achieve this, we propose *P-compositionality* (Definition 6), a *new partitioning scheme* of which Herlihy and Wing’s locality principle [1] is an instance. Recall that locality says that a concurrent system  $Q$  is linearizable if and only if each concurrent object in  $Q$  is linearizable. The crux of *P-compositionality* is that it generalizes the idea behind the locality principle to operations on the same concurrent object. For example, the operations on a concurrent unordered set and map are linearizable if and only if the *restriction to each key* is linearizable. This is not a consequence of Herlihy and Wing’s locality principle.

In this paper, we study the pragmatics of *P-compositionality* through its implementation in a novel linearizability checker and experimental evaluation. Our implementation is based on Wing and Gong’s algorithm (*WG algorithm*) [4] and a recent extension by Lowe [5]. We call Lowe’s extension of Wing and Gong’s algorithm the *WGL algorithm*. The idea behind the WGL algorithm is to prune states that are equivalent to an already seen state. Lowe’s experiments show that the WGL algorithm can solve a significantly larger number of problem instances than the WG algorithm. We therefore use the more recent WGL algorithm as our starting point.

Our linearizability checker preserves three practical properties of the algorithms in the WG-family that we deem important. Firstly, our tool is precise, i.e., it reports no false alarms. This is particularly significant for evaluating large code bases, as effectively shown by the Jepsen project. Secondly, our tool takes as input an *executable specification* of the data type to be checked. This significantly simplifies the task of expressing the expected behaviour of a data type because one merely writes code, i.e., no expertise in formal modeling is required. Finally, our tool can be easily integrated with a range of runtime monitors to generate a history from a run of a concurrent system. This is essential to make it a viable runtime verification technique.

We experimentally evaluate our linearizability checker using nine different implementations of concurrent sets, including Intel’s TBB library, as exemplars of *P-compositionality*. Our experiments show that our linearizability checker is at least one order of magnitude faster and/or more space efficient than the WGL algorithm. Overall, the results of our work can therefore dramatically increase the number of runs that can be checked for linearizability bugs in a given time budget.

The rest of this paper is organized as follows. We first formalize the problem by recalling familiar concepts (§ 2). We then present *P-compositionality* (§ 3) on which our decision procedure (§ 4) is based. We implement and experimentally evaluate our decision procedure (§ 5). Finally, we discuss related work (§ 6) and conclude the paper (§ 7).

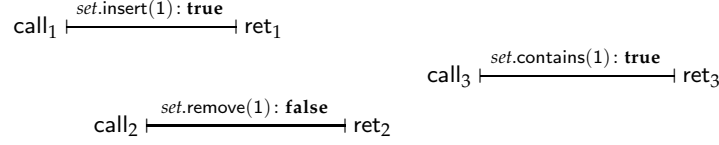


Fig. 1: A history diagram  $H_1$  for the operations on a concurrent set

## 2 Background

We recall familiar concepts that are fundamental to everything that follows.

**Definition 1 (History).** Let  $E \triangleq \{\text{call}, \text{ret}\} \times \mathbb{N}$ . For all natural numbers  $n$  in  $\mathbb{N}$ ,  $\text{call}_n \triangleq \langle \text{call}, n \rangle$  in  $E$  is called a **call** and  $\text{ret}_n \triangleq \langle \text{ret}, n \rangle$  in  $E$  is called a **return**. The invocation of a procedure with input and output arguments is called an **operation**. An **object** comprises a finite set of such operations. For all  $e$  in  $E$ ,  $\text{obj}(e)$  and  $\text{op}(e)$  denote the object and operation of  $e$ , respectively. A **history** is a tuple  $\langle H, \text{obj}, \text{op} \rangle$  where  $H$  is a finite sequence of calls and returns, totally ordered by  $\preceq_H$ . When no ambiguity arises, we simply write  $H$  for a history. We write  $|H|$  for the **length** of  $H$ .

Intuitively, a history  $H$  records a particular run of a concurrent system. Using the implicitly associated functions  $\text{obj}$  and  $\text{op}$ , a history  $H$  gives relevant information on all operations performed at runtime, and the sequence of calls and returns in  $H$  give the relative points in time at which an operation started and completed with respect to other operations. This can be visualized using the familiar history diagrams [1], as illustrated next.

*Example 1.* Consider a concurrent set with the usual operations: ‘insert’ adds an element to a set, whereas ‘remove’ does the opposite, and ‘contains’ checks membership. The return value indicates the success of the operation. For example, ‘ $\text{set.remove}(1)$ : **true**’ denotes the operation that successfully removed ‘1’ from the object ‘ $\text{set}$ ’, whereas ‘ $\text{set.remove}(1)$ : **false**’ denotes the operation that did not modify ‘ $\text{set}$ ’ because ‘1’ is already not in the set. Then the history diagram in Fig. 1 can be defined by  $H_1 = \langle \text{call}_1, \text{call}_2, \text{ret}_1, \text{ret}_2, \text{call}_3, \text{ret}_3 \rangle$  such that, for all  $1 \leq i \leq 3$ ,  $\text{obj}(\text{call}_i) = \text{obj}(\text{ret}_i) = \text{'set'}$ , and the following holds:

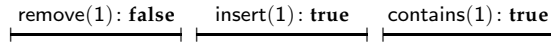
- $\text{op}(\text{call}_1) = \text{op}(\text{ret}_1) = \text{'insert}(1): \text{true}'$ ,
- $\text{op}(\text{call}_2) = \text{op}(\text{ret}_2) = \text{'remove}(1): \text{false}'$ ,
- $\text{op}(\text{call}_3) = \text{op}(\text{ret}_3) = \text{'contains}(1): \text{true}'$ .

Note that  $|H_1| = 6$  and the total ordering  $\preceq_{H_1}$  satisfies, among other constraints,  $\text{ret}_1 \preceq_{H_1} \text{call}_3$  because  $\text{ret}_1$  precedes  $\text{call}_3$  in the sequence  $H_1$ .

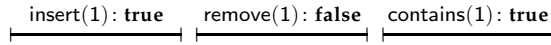
Henceforth, we draw diagrams as in Fig. 1. Linearizability is ultimately defined in terms of sequential histories, in the following sense:

**Definition 2 (Complete and sequential history).** Let  $e, e' \in E$  and  $H$  be a history. If  $e$  is a call and  $e'$  is a return in  $H$ , both are **matching** whenever  $e \preceq_H e'$  and their objects and operations are equal, i.e.  $\text{obj}(e) = \text{obj}(e')$  and  $\text{op}(e) = \text{op}(e')$ . A history is called **complete** if every call has a unique matching return. A complete history is called **sequential** whenever it alternates between matching calls and returns (necessarily starting with a call).

*Example 2.* The following history  $H_2$  is sequential:



And so is  $H_3$  that we get when we swap the first two operations in  $H_2$  (although the resulting sequence of operations is not what we would expect from a sequential set, as discussed next):



$H_3$  in Example 2 illustrates that a history can be sequential even though it may not satisfy the expected sequential behaviour of the data type. This is addressed by the following definition:

**Definition 3 (Specification).** A **specification**, denoted by  $\phi$  (possibly with a subscript), is a unary predicate on sequential histories.

*Example 3.* Define  $\phi_{\text{set}}$  to be the specification of a sequential finite set. This means that, given a sequential history  $S$  according to Definition 2, the predicate  $\phi_{\text{set}}(S)$  holds if and only if the input and output of ‘insert’, ‘remove’ and ‘contains’ in  $S$  are consistent with the operations on a set. For example,  $\phi_{\text{set}}(H_2) = \text{true}$ , whereas  $\phi_{\text{set}}(H_3) = \text{false}$  for the histories from Example 2.

*Remark 1.* In the upcoming decision procedure (§ 4), every  $\phi$  is an *executable specification*. Informally, this is achieved by ‘replaying’ all operations in a sequential history  $S$  in the order in which they appear in  $S$ . If in any step the output deviates from the expected result, the executable specification returns false; otherwise, if it reaches the end of  $S$ , it returns true.

The next definition will be key to answer which calls may be reordered in a history in order to satisfy a specification.

**Definition 4 (Happens-before).** Given a history  $H$ , the **happens-before** relation is defined to be a partial order  $<_H$  over calls  $e$  and  $e'$  such that  $e <_H e'$  whenever  $e$ ’s matching return, denoted by  $\text{ret}(e)$ , precedes  $e'$  in  $H$ , i.e.  $\text{ret}(e) \preceq_H e'$ . We say that two calls  $e$  and  $e'$  **happen concurrently** whenever  $e \not<_H e'$  and  $e' \not<_H e$ .

*Example 4.* For the history  $H_1$  in Fig. 1, we get:

- $\text{call}_1 <_{H_1} \text{call}_3$  and  $\text{call}_2 <_{H_1} \text{call}_3$ , i.e.  $\text{call}_1$  and  $\text{call}_2$  happen-before  $\text{call}_3$ ;
- $\text{call}_1 \not<_{H_1} \text{call}_2$  and  $\text{call}_2 \not<_{H_1} \text{call}_1$ , i.e.  $\text{call}_1$  and  $\text{call}_2$  happen concurrently.

Note that a history  $H$  is sequential if and only if  $<_H$  is a total order. More generally,  $<_H$  is an interval order [6]: for every  $x, y, u, v$  in  $H$ , if  $x <_H y$  and  $u <_H v$ , then  $x <_H v$  or  $u <_H y$ . Observe that a partial order  $\langle P, \leq \rangle$  is an interval order if and only if no restriction of  $\langle P, \leq \rangle$  is isomorphic to the following Hasse diagram [7]:



Put differently, this paper is about a decision procedure (§ 4) that concerns a certain class of partial orders. The decision problem rests on the next definition:

**Definition 5 (Linearizability).** Let  $\phi$  be a specification. A  $\phi$ -*sequential history* is a sequential history  $H$  that satisfies  $\phi(H)$ . A history  $H$  is **linearizable with respect to  $\phi$**  if it can be extended to a complete history  $H'$  (by appending zero or more returns) and there is a  $\phi$ -sequential history  $S$  with the same obj and op functions as  $H'$  such that

**L1**  $H'$  and  $S$  are equal when seen as two sets of calls and returns;

**L2**  $<_H \subseteq <_S$ , i.e. for all calls  $e, e'$  in  $H$ , if  $e$  happens-before  $e'$ , the same is true in  $S$ .

Informally, extending  $H$  to  $H'$  means that all pending operations have completed. This paper therefore considers only complete histories. This is fully justified under our stated assumption (§ 1) that the concurrent system is deadlock-free [5]. Condition **L1** means that  $H'$  and  $S$  are identical if we disregard the order in which calls and returns occur in both sequences. Condition **L2** says that the happens-before relation between calls in  $H$  must be preserved in  $S$ .

*Example 5.* Recall Example 3. Then  $H_1$  in Fig. 1 is linearizable with respect to  $\phi_{set}$  because  $H_2$  is a witness for a  $\phi_{set}$ -sequential history that respects the happens-before relation  $<_{H_1}$  detailed in Example 4. In particular,  $\text{call}_1 <_{H_1} \text{call}_3$  and  $\text{call}_2 <_{H_1} \text{call}_3$  cannot be reordered.

### 3 $P$ -compositionality

In this section, we introduce  $P$ -compositionality. We illustrate our new partitioning scheme in Examples 7–9.

**Definition 6 ( $P$ -compositionality).** Let  $P$  be a function that maps a history  $H$  to a non-trivial partition of  $H$ , i.e.  $P$  satisfies  $P(H) \neq \{H\}$ . A specification  $\phi$  is called  **$P$ -compositional** whenever any history  $H$  is linearizable with respect to  $\phi$  if and only if, for every history  $H' \in P(H)$ ,  $H'$  is linearizable with respect to  $\phi$ . When this equivalence holds we speak of  **$P$ -compositionality**.

In the following examples, we assume that the partitions are non-trivial. The first example illustrates that the locality principle [1] is an instance of  $P$ -compositionality.

*Example 6.* Denote with  $Obj$  the set of objects. Let  $\phi$  be a specification for all objects in  $Obj$ . Let  $P_{Obj}$  be the function that maps every history  $H$  to the set of histories  $\mathcal{H}$  where each sub-history  $H' \in \mathcal{H}$  is the restriction of  $H$  to an object in  $Obj$ . Then  $P_{Obj}(H)$  is a partition of  $H$ . By the locality principle [1], a history  $H$  is linearizable with respect to  $\phi$  if and only if, for all  $H_{obj} \in P_{Obj}(H)$ ,  $H_{obj}$  is linearizable with respect to  $\phi$ . Therefore  $\phi$  is a  $P_{Obj}$ -compositional specification.

The remaining examples show that  $P$ -compositionality strictly generalizes the locality principle because  $P$ -compositionality can partition a history even if the implementation details or constituent parts (i.e. objects) of a concurrent system are unknown. For example, there are at least eight different implementations of concurrent sets (Table 2), but we do not need to know the objects (e.g. registers, buckets) of which such implementations consist in order to partition one of their histories. This is in contrast to the locality principle where such knowledge is required. Put differently,  $P$ -compositionality is all about the *interface* of a concurrent data type, whereas the locality principle hinges on the *implementation details* of such an interface.

*Example 7.* Reconsider  $\phi_{set}$ , the specification of a set from Example 3, where all operations have the form  $insert(k)$ ,  $remove(k)$  and  $contains(k)$  for some  $k$ . Let  $P_{set}$  be the function that partitions every history  $H$  according to such  $k$ . Since the ‘insert’, ‘remove’ and ‘contains’ operations on a single set object are linearizable if and only if the restriction to each  $k$  is linearizable,  $\phi_{set}$  is a  $P_{set}$ -compositional specification of a set.

Similarly, there exists a  $P_{map}$ -compositional specification for concurrent unordered maps where every history is partitioned by each key  $k$ .

*Example 8.* Consider a concurrent array. As their sequential counterparts, a concurrent array can be only read or written at a particular array index. Let  $P_{array}$  be the function that partitions a history based on such array indexes. This gives a  $P_{array}$ -compositional specification of an array.

*Example 9.* Consider a concurrent stack where each pop and push operation also returns the height of the stack before it is modified. Among other things, the return value can be used to determine whether the operation has succeeded. For example, if  $stack.pop$  returns zero, we know the pop operation was unsuccessful (and the popped element is undefined) because the stack was empty at the time the operation was called. We can use the returned height to partition a history such that a concurrent stack is linearizable if and only if each partition is linearizable. This way we get a  $P_{stack}$ -compositional specification of a stack.

Intuitively, the reason why the previous specifications are  $P$ -compositional is because all operations in one partition are, informally speaking, unaffected by all operations in every other partition. For example, the return value of  $set.insert(k)$  is unaffected by  $set.insert(k')$ ,  $set.remove(k')$  and  $set.contains(k')$  for  $k \neq k'$ . This clearly, however, has its limitations. For example, a ‘size’ operation that returns the number of elements in a concurrent collection data type cannot be generally partitioned this way.

Note that all these examples have in common that their  $P$ -compositional specifications can be expressed as a conjunction of specifications that each partition a history. For example,  $\phi_{set} = \bigwedge_{k \in K} \phi_{set(k)}$  where  $\phi_{set(k)}$  for every  $k$  is a sequential specification that only concerns operations on  $k$ , e.g.  $set.insert(k)$ .

Next, we show how to leverage the concept of  $P$ -compositional to more efficiently find linearizability bugs.

## 4 Decision procedure

In this section, we explain our linearizability checking algorithm that decides whether a history is linearizable with respect to some  $P$ -compositional specification (Definition 6). The novelty of our decision procedure is Algorithm 3 that leverages  $P$ -compositional. In the next section (§ 5), we experimentally evaluate the effectiveness of Algorithm 3.

Since we base our work on the WGL algorithm (recall § 1), we use the following data structures to represent the input to the decision procedure:

1. The specification (Definition 3) is modelled by a persistent data structure, e.g. [8]. Most standard data types in functional programming languages can be almost directly used this way. For instance, the specification of a set can be modelled through an immutable sequential set.
2. A history (Definition 1), in turn, is represented by a doubly-linked list of so-called **entries**. Consequently, each entry  $e$  has a  $e.next$  and  $e.prev$  field that point to the next and previous entry, respectively. In addition, each entry  $e$  has a `match` field, and we say that  $e$  is a **call entry** exactly if  $e.match \neq \text{null}$ ; otherwise,  $e$  is called a **return entry**. Given a call entry  $e$ ,  $e.match$  corresponds to the **matching return entry** of  $e$ . This linked-list data structure therefore aligns directly with the usual definition of history (Definition 1).

The idea behind the WGL Algorithm 1 is threefold: it keeps track of provisionally linearized call entries in a stack; it uses the stack to backtrack if necessary, and caches already seen configurations. We briefly explain each idea in turn. Denote the stack of call entries by `calls`. Given a history  $H$ , the height of `calls` is at most half of  $H$ 's length, i.e.  $|\text{calls}| \leq 0.5 \times |H| = N$ . Note that there is no rounding involved because  $|H|$  is always even since every call entry has a matching return entry. The height of the stack grows only if a call entry can be linearized (line 5). When the stack grows or shrinks, the history is modified (lines 13 and 23) by the `LIFT` and `UNLIFT` procedures (Algorithm 2). We remark that the workings of both procedures are illustrated by Example 10. If no further call entries can be linearized but the stack is nonempty, the algorithm backtracks and tries the next possible call entry (lines 18–24). The backtracking points depend on the return value of `apply(entry, s)` and the cache. The former (line 3) models the specification  $\phi$ : by Remark 1, it determines whether entry can be applied to the current state  $s$  of a persistent data type. The latter (lines 4–8) is an optimization due to Lowe [5] that prunes the search space by memoizing already seen configurations which are known to be non-linearizable. More



---

**Algorithm 1** WGL linearizability checker [5]

---

**Require:** `head_entry` is such that `head_entry.next` points to the beginning of history  $H$ .  
**Require:**  $N = 0.5 \times |H|$  is half of the total number of entries reachable from `head_entry`.  
**Require:** `linearized` is a bitset (array of bits) such that `linearized[k] = 0` for all  $0 \leq k < N$ .  
**Require:** For all entries  $e$  in  $H$ ,  $0 \leq \text{entry\_id}(e) < N$ .  
**Require:** For all entries  $e$  and  $e'$  in  $H$ , if  $\text{entry\_id}(e) = \text{entry\_id}(e')$ , then  $e = e'$ .  
**Require:** `cache` is an empty set and `calls` is an empty stack.

```
1: while head_entry.next  $\neq$  null do
2:   if entry.match  $\neq$  null then                                      $\triangleright$  Is call entry?
3:      $\langle \text{is\_linearizable}, s' \rangle \leftarrow \text{apply}(\text{entry}, s)$        $\triangleright$  Simulate entry's operation
4:     cache'  $\leftarrow$  cache                                            $\triangleright$  Copy set
5:     if is_linearizable then
6:       linearized'  $\leftarrow$  linearized                                 $\triangleright$  Copy bitset
7:       linearized'[entry_id(entry)]  $\leftarrow$  1                       $\triangleright$  Insert entry_id(entry) into bitset
8:       cache  $\leftarrow$  cache  $\cup \{ \langle \text{linearized}', s' \rangle \}$        $\triangleright$  Update configuration cache
9:     if cache'  $\neq$  cache then
10:      calls  $\leftarrow \text{push}(\text{calls}, \langle \text{entry}, s \rangle)$   $\triangleright$  Provisionally linearize call entry and state
11:      s  $\leftarrow$  s'                                                   $\triangleright$  Update state of persistent data type
12:      linearized[entry_id(entry)]  $\leftarrow$  1                         $\triangleright$  Keep track of linearized entries
13:      LIFT(entry)                                                   $\triangleright$  Provisionally remove the entry from the history
14:      entry  $\leftarrow$  head_entry.next                                $\triangleright$  Continue search in shortened history
15:    else                                                             $\triangleright$  Cannot linearize call entry
16:      entry  $\leftarrow$  entry.next                                      $\triangleright$  Continue search in unmodified history
17:  else                                                             $\triangleright$  Handle "return entry"
18:    if is_empty(calls) then
19:      return false                                                 $\triangleright$  Cannot linearize entries in history
20:       $\langle \text{entry}, s \rangle \leftarrow \text{top}(\text{calls})$                    $\triangleright$  Revert to earlier state
21:      linearized[entry_id(entry)]  $\leftarrow$  0
22:      calls  $\leftarrow \text{pop}(\text{calls})$ 
23:      UNLIFT(entry)                                                 $\triangleright$  Undo provisional linearization
24:      entry  $\leftarrow$  entry.next
25: return true
```

---

accurately, each configuration is a pair that consists of a set of unique call entry identifiers and a state of the persistent data structure. The intuition behind pruning already seen configurations is that only one of two permutations of operations on a concurrent data type need to be considered if they lead to an identical state [5]. We remark that the total correctness of the WGL algorithm follows from Wing and Gong's total correctness argument [4].

*Example 10.* We illustrate the handling of entries in the history data structure. For this, consider the two histories in Fig. 2. In Fig. 2a, the entries satisfy the following: `call2.prev = call1`, `call2.next = call3` and `call2.match = ret2` etc. Then `LIFT(call2)` (Algorithm 2) produces the history shown in Fig. 2b. Note that both `call2` and `ret2` are still valid entry pointers whose fields remain unchanged. This explains how `UNLIFT(call2)` reverts the change in constant-time.



Algorithm 3 gives our partitioning scheme. This is an iterative algorithm that, given an entry in a history  $H$  and positive integer  $n$ , partitions  $H$  starting from that entry into at most  $n$  separate sub-histories. The partitioning is controlled by the function  $partition: E \rightarrow \mathbb{N}$  from the set of call and return entries to the natural numbers.

*Example 11.* Consider the history in Fig. 2b. For all entries  $e$  in this history, let  $partition(e) = k$  where  $k$  is the integer argument of the operation. For example,  $partition(call_3) = partition(ret_3) = 1$  because  $op(call_3) = op(ret_3) = \text{'remove(1) : false'}$ . Then the function  $PARTITION(call_1)$  returns two disjoint sub-histories for the operations on '0' and '1', respectively:

$$\begin{array}{ccc} call_1 \xrightarrow{\text{set.insert(0) : true}} ret_1 & \text{and} & call_3 \xrightarrow{\text{set.remove(1) : false}} ret_3. \\ & & call_2 \xrightarrow{\text{set.contains(0) : true}} ret_2 \end{array}$$

Given a nonempty set of disjoint sub-histories returned by the  $PARTITION$  function (Algorithm 3), we invoke Algorithm 1 on each sub-history. It is not too difficult to implement sub-histories such that there is no sharing between them, and Algorithm 1 could be therefore run in parallel for each sub-history. Nevertheless, this addresses a challenging problem that was identified independently by Lowe [5] and Kingsbury [9].

**Theorem 1.** *Let  $\phi$  be a  $P$ -compositional specification and  $H$  be a history. Denote with  $head\_entry$  the entry that represents the beginning of  $H$ . Associate with each disjoint history  $H_k$  in partition  $P(H)$  a unique number  $0 \leq k < |P(H)| = n$ . If, for all  $H_k \in P(H)$  and  $e \in H_k$ ,  $partition(e) = k$ , then  $H$  is linearizable with respect to  $\phi$  if and only if Algorithm 1 returns true for every history in  $PARTITION(head\_entry, n)$ .*

We next experimentally quantify the benefits of the previous theorem.

## 5 Implementation and experiments

In this section, we discuss and experimentally evaluate our implementation of the decision procedure (§ 5). As an exemplar of  $P$ -compositionality, our experiments use Intel's TBB library and Lowe's implementations of concurrent sets.

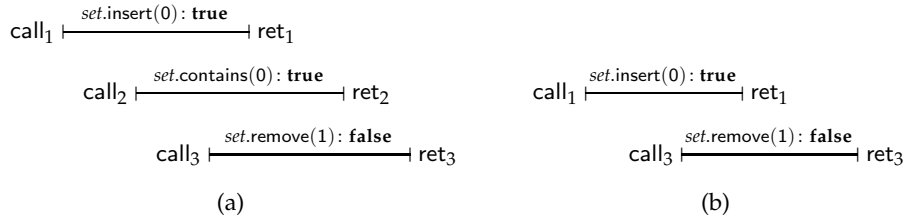


Fig. 2: After calling  $LIFT(call_2)$  in history (2a), we get the history in (2b).  $UNLIFT(call_2)$  reverts this change in constant-time.

---

**Algorithm 2** History modifications

---

```
1: procedure LIFT(entry)
2:   entry.prev.next  $\leftarrow$  entry.next
3:   entry.next.prev  $\leftarrow$  entry.prev
4:   match  $\leftarrow$  entry.match
5:   match.prev.next  $\leftarrow$  match.next
6:   if match.next  $\neq$  null then
7:     match.next.prev  $\leftarrow$  match.prev
8:
9: procedure UNLIFT(entry)
10:  match  $\leftarrow$  entry.match
11:  match.prev.next  $\leftarrow$  match
12:  if match.next  $\neq$  null then
13:    match.next.prev  $\leftarrow$  match
14:  entry.prev.next  $\leftarrow$  entry
15:  entry.next.prev  $\leftarrow$  entry
```

---

---

**Algorithm 3** History partitioner

---

```
Require:  $n$  is a positive integer
Require: entries is an array of size  $n$ 
1: function PARTITION(entry,  $n$ )
2:   for  $0 \leq i < n$  do
3:     entries[ $i$ ]  $\leftarrow$  null
4:   while entry  $\neq$  null do
5:      $i \leftarrow \text{partition}(\text{entry}) \bmod n$ 
6:     if entries[ $i$ ]  $\neq$  null then
7:       entries[ $i$ ].next  $\leftarrow$  entry
8:     next_entry  $\leftarrow$  entry.next
9:     entry.prev  $\leftarrow$  entries[ $i$ ]
10:    entry.next  $\leftarrow$  null
11:    entries[ $i$ ]  $\leftarrow$  entry
12:    entry  $\leftarrow$  next_entry
13: return entries
```

---

## 5.1 Implementation

The implementation details of an NP-complete decision procedure matter, especially for our experimental evaluation of  $P$ -compositionality. We particularly consider hashing and cache eviction options because these were not studied in previous implementations of the WG-based algorithms [4,5].

For experimental robustness, we implemented our linearizability checker in C++11 [10] because this language has built-in concurrency support while allowing us to rule out interference from managed runtime environments (e.g. JVM) due to garbage collection etc. The choice of language, though, meant that we had to implement persistent data structures from scratch. In doing so, we focused on optimizing equality checks for our specific purposes. This way, we managed to avoid a known performance bottleneck in Lowe’s implementation of the WGL algorithm [5] where the cost of equality checks had to be compensated with an additional union-find data structure. Another optimization in our implementation is a constant-time (instead of linear-time) hash function for bitsets where we exploit the fact that the bitwise XOR operator over fixed-size bit vectors forms an abelian group. This optimization turns out to be important when histories are longer than 8 K, cf. [5]. To see this, consider the computational steps for retrieving a configuration from the cache and updating it (line 8 in Algorithm 1). For example, a history of length  $2^{16}$  means that each bitset in a configuration is at least 3 KiB, and so a constant-time hash function can make a measurable difference when the cache is frequently accessed. In fact, it is not uncommon for the cache to contain more than 27 K of such configurations. For this reason, we also implemented a *least recently used* (LRU) cache eviction feature that can optionally be enabled at compile-time. The effects of the LRU cache will be evaluated shortly.

Benchmark	WGL			WGL+LRU			WGL+P		
	Time	Memory	Timeout	Time	Memory	Timeout	Time	Memory	Timeout
TBB	101 s	9792 MiB	0%	11 s	<b>670 MiB</b>	0%	<b>6 s</b>	672 MiB	0%
CRLSL	20 s	15738 MiB	0%	25 s	678 MiB	0%	<b>6 s</b>	<b>400 MiB</b>	0%
CRLFSL	14 s	15029 MiB	0%	18 s	678 MiB	0%	<b>5 s</b>	<b>401 MiB</b>	0%
FGL	16 s	14297 MiB	0%	81 s	678 MiB	0%	<b>5 s</b>	<b>401 MiB</b>	0%
LLL	23 s	16494 MiB	0%	94 s	678 MiB	0%	<b>6 s</b>	<b>401 MiB</b>	0%
LSL	20 s	15736 MiB	0%	25 s	678 MiB	14%	<b>6 s</b>	<b>401 MiB</b>	0%
LFLL	11 s	11847 MiB	0%	15 s	678 MiB	0%	<b>5 s</b>	<b>402 MiB</b>	0%
LFSL	14 s	14712 MiB	0%	18 s	678 MiB	0%	<b>5 s</b>	<b>401 MiB</b>	0%
LFSLF0	14 s	13125 MiB	0%	18 s	678 MiB	0%	<b>5 s</b>	<b>402 MiB</b>	0%
LFSLF1	< 1 s	404 MiB	0%	< 1 s	407 MiB	0%	< 1 s	402 MiB	0%
OPTIMIST	16 s	13818 MiB	0%	54 s	678 MiB	9%	<b>5 s</b>	<b>401 MiB</b>	0%

Table 1: Experimental results for three variants of the same linearizability checker. The results for the baseline are reported in the WGL column. The rows correspond to benchmarks drawn from Intel’s TBB library and Lowe’s implementations of concurrent sets (see Table 2 for mnemonics).

Overall, our implementation and experimental setup is around 4 K lines of code, including several dozen unit tests. All the code and benchmarks are publicly available in our source code repository.<sup>2</sup>

## 5.2 TBB and concurrent set experiments

For the experimental evaluation of our partitioning scheme, we collected over 700 histories from nine different implementations of concurrent sets by Lowe [5] and the concurrent unordered set implementation in Intel’s TBB library.<sup>3</sup> We performed all experiments on a 64-bit machine running GNU/Linux 3.17 with 12 Intel Xeon 2.4 GHz cores and 94 GB of main memory.

Each history is generated by running 4 concurrent threads that pseudo randomly invoke operations on a single shared concurrent set. The argument of each operation is a pseudo random uniformly distributed integer between 0 (inclusive) and 24 (exclusive). Each thread invokes 70 K such operations. Note that this is significantly more than in previous experiments where each process is limited to  $2^{13} \approx 8$  K operations [5]. In total, since every call generates a pair of entries, every history  $H$  in our benchmarks has length  $|H| = 4 \times 2 \times 70 \text{ K} = 560 \text{ K}$ . We discuss the experimental results using Intel’s TBB library and Lowe’s concurrent set implementations in turn.

<sup>2</sup> <https://github.com/ahorn/linearizability-checker>

<sup>3</sup> <https://www.threadingbuildingblocks.org/>

The experimental results are given in Table 1. Each of the three main columns corresponds to one variant of the same linearizability checker: ‘WGL’ is the baseline, ‘WGL+LRU’ is the WGL algorithm with LRU cache eviction enabled (§ 5.1), and ‘WGL+P’ is the WGL algorithm combined with our partitioning algorithm (Algorithm 3 in § 4). We tried to use the WG algorithm [4] without the extension by Lowe [5] but WG times out on the majority of benchmarks. We therefore do not report the results on the WG algorithm and focus on WGL, WGL+LRU and WGL+P. The meaning of the sub-columns is as follows. The ‘Time’ and ‘Memory’ columns give the average of the elapsed time and virtual memory usage, respectively. These averages exclude runs that we had to terminate after 1 hour. The percentage of such terminated runs is given in the ‘Timeout’ column. In each row, all variants are compared with respect to the same benchmark data. We therefore do not report confidence intervals.

The TBB benchmark corresponds to the first row in Table 1 and consists of a total of 100 histories. Table 1 clearly shows that the WGL+P algorithm is at least one order of magnitude faster compared to the baseline. We also see that enabling the LRU cache eviction decreases the memory footprint by at least one order of magnitude, approximately 10 GiB versus 700 MiB. In fact, the runtime performance of WGL+LRU is almost one order of magnitude faster than the baseline. The WGL+P algorithm is at least as fast and almost as space efficient as WGL+LRU. In the experiments with Lowe’s implementations of concurrent sets (see next paragraph), we further investigate the effect of the LRU cache eviction feature and how it compares to the partitioning scheme.

We give Lowe’s implementations of concurrent sets mnemonics (Table 2) that identify the remaining ten benchmarks in Table 1. Each of these ten benchmarks comprises between 50 and 100 histories with an average of 70 histories per benchmark. To avoid bias, we collected these using Lowe’s tool. The significance of the experimental results in Table 1 is twofold. Firstly, they show that on average, WGL+P is three times faster than WGL, and WGL+P consumes one order of magnitude less space than WGL. Secondly, and more crucially, however, these experiments reveal that WGL+LRU is not as efficient as WGL+P, in neither time nor space. For example, for WGL+LRU the average elapsed time of the FGL and LLL benchmark is 81 s and 94 s, respectively, with an average memory usage of 678 MiB in both cases. By contrast, WGL+P achieves an average runtime of less than 7 s (and so WGL+P is one order of magnitude faster than WGL+LRU) and consumes even less memory on average (401 MiB) than WGL+LRU. The higher average runtime of WGL+LRU in the FGL benchmark is due to a single check that took several orders of magnitude longer (3068 s) than the remaining checks (20 s on average when the 3068 s outlier is excluded). In the LLL benchmark there are two such outliers (2201 s and 675 s, whereas the other checks average 27 s). The observed difference between WGL+LRU and WGL+P is even more pronounced in both the LSL and OPTIMIST benchmarks where the LRU cache eviction causes 14% and 9% of runs to timeout, whereas the WGL+P algorithm always runs to completion in less than a few seconds.

Benchmark name	Mnemonic	Benchmark name	Mnemonic
collision resistance lazy skip list	CRLSL	lock-free linked-list	LFLL
collision resistance lock-free skip list	CRLFSL	lock-free skip list	LFSL
fine-grained lock	FGL	lock-free skip list faulty (bad hash)	LFSLF0
lazy linked-list	LLL	lock-free skip list faulty (good hash)	LFSLF1
lazy skip list	LSL	optimistic lock	OPTIMIST

Table 2: Mnemonics for Lowe’s implementation of concurrent sets [5]

This experimentally confirms that the WGL+P is one order of magnitude faster as well as more space efficient than the baseline and WGL+P consumes even less space than our WGL+LRU implementation.

## 6 Related work

Linearizability is related to the concept of atomicity, including weaker forms such as  $k$ -atomicity [11]. An important difference is that atomicity is typically not defined in terms of a sequential specification, e.g. [12]. The theoretical limitations of automatically verifying linearizability are well understood. Of course, the problem is generally undecidable [13]. In fact, even checking finite-state implementation against atomic specifications, provided the number of program threads is bounded, is EXPSPACE [14]. And the best known lower bound for this problem is PSPACE-hardness. This explains the restrictions in this paper and its focus on runtime verification instead.

The literature on machine-assisted techniques for checking linearizability can be broadly divided into simulation-based methods (e.g. [15,16]), model checking (e.g. [17,18,19,20]), static analysis (e.g. [21,22,23,24]) and fully automatic testing (e.g. [4,25,26,27,28,29,30,5]). The simulation-based methods have been used by experts to mechanically verify simple fine-grained and lock-free implementations. Model checking requires less expertise but is typically limited to very small programs and a small number of threads due to the state explosion problem. By contrast, static analysis tools aim to prove correctness with respect to an unbounded number of threads. In general, these techniques are necessarily incomplete and require the user to supply linearization points and/or invariants. Vafeiadis [24] proposes a more automatic form of static analysis that works well on simpler concurrent data types such as stacks but reportedly not so well on data types that have more complicated invariants, including the CAS-based and lazy concurrent sets extensively studied in our experiments.

Our work is most closely related to linearizability testing techniques that are precise, fully automatic and necessarily incomplete, e.g. [4,25,26,27,28,29,30,5]. We focus our discussion on tools that do not require the notion of commit points, cf. [31]. The work in [25,30] checks  $k$ -atomicity with a polynomial-time algorithm assuming that each write to a register assigns a distinct value. By contrast, we solve a more general NP-complete problem of which  $k$ -atomicity is an instance. The tool in [26] analyzes code that uses concurrent collection data types such as maps. To make the analysis scale, the authors assume that the collection data types are linearizable, whereas our tool could be used to check such

an assumption. A different tool [27] requires programmers to annotate concurrent implementations with so-called state summary functions that act as a form of specification. Our approach is more modular because it strictly separates the concurrent implementation from its specification. By contrast, [28] works without the programmer having to provide a sequential specification. As a result, however, the tool can only find linearizability violations when an exception is thrown or a deadlock occurs. Subsequent work [29] circumvents this, in the context of object-oriented programs, by considering the special case of a superclass serving as an executable, possibly non-deterministic, specification for all its subclasses. The fact that the superclass can be non-deterministic may explain why even checks of two threads can take a significant amount of time (e.g. 108 min) despite the fact that each concurrent test considers only two possible linearizations [29]. By contrast, the WGL algorithm [4,5], on which our decision procedure is based (§ 4), is significantly faster but limited to deterministic specifications. Crucially, our experiments (§ 5) with  $P$ -compositional specifications show a significant improvement over the WGL algorithm.

## 7 Concluding remarks

We have presented a precise, fully automatic runtime verification technique for finding linearizability bugs in implementations of concurrent data types that are expected to satisfy a  $P$ -compositional specification. Our experiments show that our partitioning scheme improves the WGL algorithm [4,5] by one order of magnitude, in both time and space. An additional strength of our technique is that it is applicable to any linearizability checker. For this, however, our work assumes that the specification is  $P$ -compositional. This is generally not always the case and it would be therefore interesting to further generalize  $P$ -compositionality, perhaps with a less modular partitioning scheme that can make more assumptions about the underlying decision procedure.

*Acknowledgements.* We would like to thank Gavin Lowe, Kyle Kingsbury and Alexey Gotsman for invaluable discussions.

## References

1. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3) (July 1990) 463–492
2. Gilbert, S., Lynch, N.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* **33**(2) (June 2002) 51–59
3. Gibbons, P.B., Korach, E.: Testing shared memories. *SIAM J. Comput.* **26**(4) (August 1997) 1208–1244
4. Wing, J.M., Gong, C.: Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.* **17**(1-2) (January 1993) 164–182
5. Lowe, G.: Testing for linearizability. In: *PODC’15*. (2015) Under submission. <http://www.cs.ox.ac.uk/people/gavin.lowe/LinearizabilityTesting/>.
6. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: *POPL’15*, ACM (2015) 651–662

7. Rabinovitch, I.: The dimension of semiorders. *Journal of Combinatorial Theory, Series A* **25**(1) (1978) 50–61
8. Okasaki, C.: *Purely Functional Data Structures*. Cambridge University Press (1998)
9. Kingsbury, K.: Computational techniques in Knossos. <https://aphyr.com/posts/314-computational-techniques-in-knossos> (May 2014)
10. ISO: International Standard ISO/IEC 14882:2011(E) Programming Language C++. International Organization for Standardization (2011)
11. Aiyer, A., Alvisi, L., Bazzi, R.A.: On the availability of non-strict quorum systems. In: *Proceedings of the 19th International Conference on Distributed Computing, DISC'05*, Springer (2005) 48–62
12. Wang, L., Stoller, S.D.: Static analysis of atomicity for programs with non-blocking synchronization. In: *PPoPP '05*, ACM (2005) 61–71
13. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: *ESOP'13*, Springer (2013) 290–309
14. Alur, R., McMillan, K., Peled, D.: Model-checking of correctness conditions for concurrent objects. *Inf. Comput.* **160**(1-2) (July 2000) 167–188
15. Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. *Electron. Notes Theor. Comput. Sci.* **137**(2) (2005) 93–110
16. Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.* **33**(1) (January 2011) 4:1–4:43
17. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: *SPIN'09*, Springer (2009) 261–278
18. Burckhardt, S., Dorn, C., Musuvathi, M., Tan, R.: Line-up: A complete and automatic linearizability checker. *SIGPLAN Not.* **45**(6) (June 2010) 330–340
19. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: *CAV'10*. (2010) 465–479
20. Liu, Y., Chen, W., Liu, Y.A., Sun, J., Zhang, S.J., Dong, J.S.: Verifying linearizability via optimized refinement checking. *IEEE Trans. Softw. Eng.* **39**(7) (2013) 1018–1039
21. Amit, D., Rinetzk, N., Repts, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: *CAV'07*, Springer (2007) 477–490
22. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: *CAV '08*, Springer (2008) 399–413
23. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: *VMCAI '09*, Springer (2009) 335–348
24. Vafeiadis, V.: Automatically proving linearizability. In: *CAV'10*, Springer (2010) 450–464
25. Anderson, E., Li, X., Shah, M.A., Tucek, J., Wylie, J.J.: What consistency does your key-value store actually provide? *HotDep'10*, USENIX Association (2010) 1–16
26. Shacham, O., Bronson, N., Aiken, A., Sagiv, M., Vechev, M., Yahav, E.: Testing atomicity of composed concurrent operations. *SIGPLAN Not.* **46**(10) (October 2011) 51–64
27. Fonseca, P., Li, C., Rodrigues, R.: Finding complex concurrency bugs in large multi-threaded applications. In: *EuroSys '11*, ACM (2011) 215–228
28. Pradel, M., Gross, T.R.: Fully automatic and precise detection of thread safety violations. *SIGPLAN Not.* **47**(6) (June 2012) 521–530
29. Pradel, M., Gross, T.R.: Automatic testing of sequential and concurrent substitutability. In: *ICSE'13*, IEEE Press (2013) 282–291
30. Golab, W., Hurwitz, J., Li, X.S.: On the k-atomicity-verification problem. *ICDCS '13*, IEEE Computer Society (2013) 591–600
31. Elmas, T., Tasiran, S., Qadeer, S.: VYRD: Verifying concurrent programs by runtime refinement-violation detection. *SIGPLAN Not.* **40**(6) (June 2005) 27–37