Local Linearizability for Concurrent Container-Type Data Structures*

Andreas Haas[1], Thomas A. Henzinger[2], Andreas Holzer[3], Christoph M. Kirsch[4], Michael Lippautz[5], Hannes Payer[6], Ali Sezgin[7], Ana Sokolova[8], and Helmut Veith[9]

1 **Google Inc.**
2 **IST Austria, Austria**
3 **University of Toronto, Canada**
4 **University of Salzburg, Austria**
5 **Google Inc.**
6 **Google Inc.**
7 **University of Cambridge, UK**
8 **University of Salzburg, Austria**
9 **Vienna University of Technology, Austria and**
Forever in our hearts

Abstract

The semantics of concurrent data structures is usually given by a sequential specification and a consistency condition. Linearizability is the most popular consistency condition due to its simplicity and general applicability. Nevertheless, for applications that do not require all guarantees offered by linearizability, recent research has focused on improving performance and scalability of concurrent data structures by relaxing their semantics.

In this paper, we present local linearizability, a relaxed consistency condition that is applicable to *container-type* concurrent data structures like pools, queues, and stacks. While linearizability requires that the effect of each operation is observed by all threads at the same time, local linearizability only requires that for each thread T, the effects of its local insertion operations and the effects of those removal operations that remove values inserted by T are observed by all threads at the same time. We investigate theoretical and practical properties of local linearizability and its relationship to many existing consistency conditions. We present a generic implementation method for locally linearizable data structures that uses existing linearizable data structures as building blocks. Our implementations show performance and scalability improvements over the original building blocks and outperform the fastest existing container-type implementations.

1998 ACM Subject Classification D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics; E.1 [Data Structures]: Lists, stacks, and queues; D.1.3 [Software]: Programming Techniques—Concurrent Programming

Keywords and phrases (concurrent) data structures, relaxed semantics, linearizability

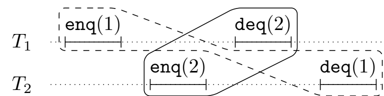
Digital Object Identifier 10.4230/LIPIcs.CONCUR.2016.6

* This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund (FWF): S11402-N23, S11403-N23, S11404-N23, S11411-N23), a Google PhD Fellowship, an Erwin Schrödinger Fellowship (Austrian Science Fund (FWF): J3696-N26), EPSRC grants EP/H005633/1 and EP/K008528/1, the Vienna Science and Technology Fund (WWTF) trough grant PROSEED, the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grant Z211-N23 (Wittgenstein Award).

© Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith;
licensed under Creative Commons License CC-BY

27th International Conference on Concurrency Theory (CONCUR 2016).
Editors: Josée Desharnais and Radha Jagadeesan; Article No. 6; pp. 6:1–6:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



The thread-induced history of thread T_1 is enclosed by a dashed line while the thread-induced history of thread T_2 is enclosed by a solid line.

Figure 1 Local Linearizability.

1 Introduction

Concurrent data structures are pervasive all along the software stack, from operating system code to application software and beyond. Both correctness and performance are imperative for concurrent data structure implementations. Correctness is usually specified by relating concurrent executions, admitted by the implementation, with sequential executions, admitted by the sequential version of the data structure. The latter form the *sequential specification* of the data structure. This relationship is formally captured by *consistency conditions*, such as linearizability, sequential consistency, or quiescent consistency [22].

Linearizability [23] is the most accepted consistency condition for concurrent data structures due to its simplicity and general applicability. It guarantees that the effects of all operations by all threads are observed consistently. This imposes the need of extensive synchronization among threads which may in turn jeopardize performance and scalability. In order to enhance performance and scalability of implementations, recent research has explored relaxed sequential specifications [20, 35, 2], resulting in well-performing implementations of concurrent data structures [2, 16, 20, 25, 33, 6]. Except for [24], the space of alternative consistency conditions that relax linearizability has been left unexplored to a large extent. In this paper, we explore (part of) this gap by investigating *local linearizability*, a novel consistency condition that is applicable to a large class of concurrent data structures that we call *container-type* data structures, or *containers* for short. Containers include pools, queues, and stacks. A fine-grained spectrum of consistency conditions enables us to describe the semantics of concurrent implementations more precisely, e.g. in the extended version of our paper [15] we show that work stealing queues [30] which could only be proven to be linearizable wrt pool are actually locally linearizable wrt double-ended queue.

Local linearizability is a (thread-)local consistency condition that guarantees that insertions *per thread* are observed consistently. While linearizability requires a consistent view over all insertions, we only require that projections of the global history—so called *thread-induced histories*—are linearizable. The induced history of a thread T is a projection of a program execution to the insert-operations in T combined with all remove-operations that remove values inserted by T irrespective of whether they happen in T or not. Then, the program execution is locally linearizable iff each thread-induced history is linearizable. Consider the example (sequential) history depicted in Figure 1. It is not linearizable wrt a queue since the values are not dequeued in the same order as they were enqueued. However, each thread-induced history is linearizable wrt a queue and, therefore, the overall execution is locally linearizable wrt a queue. In contrast to semantic relaxations based on relaxing sequential semantics such as [20, 2], local linearizability coincides with *sequential correctness* for single-threaded histories, i.e., a single-threaded and, therefore, sequential history is locally linearizable wrt a given sequential specification if and only if it is admitted by the sequential specification.

Local linearizability is to linearizability what coherence is to sequential consistency. Coherence [19], which is almost universally accepted as the absolute minimum that a shared memory system should satisfy, is the requirement that there exists a unique global order per shared memory location. Thus, while all accesses by all threads to a given memory location have to conform to a unique order, consistent with program order, the relative ordering of accesses to multiple memory locations do not have to be the same. In other words, coherence is sequential consistency per memory location. Similarly, local linearizability is linearizability per local history. In our view, local linearizability offers enough consistency for the correctness of many applications as it is the local view of the client that often matters. For example, in a locally linearizable queue each client (thread) has the impression of using a perfect queue—no reordering will ever be observed among the values inserted by a single thread. Such guarantees suffice for many e-commerce and cloud applications. Implementations of locally linearizable data structures have been successfully applied for managing free lists in the design of the fast and scalable memory allocator scalloc [5]. Moreover, except for fairness, locally linearizable queues guarantee all properties required from Dispatch Queues [1], a common concurrency programming mechanism on mobile devices.

In this paper, we study theoretical and practical properties of local linearizability. Local linearizability is compositional—a history over multiple concurrent objects is locally linearizable iff all per-object histories are locally linearizable (see Thm. 12) and locally linearizable container-type data structures, including queues and stacks, admit only “sane” behaviours—no duplicated values, no values returned from thin air, and no values lost (see Prop. 4). Local linearizability is a weakening of linearizability for a natural class of data structures including pools, queues, and stacks (see Sec. 4). We compare local linearizability to linearizability, sequential, and quiescent consistency, and to many shared-memory consistency conditions.

Finally, local linearizability leads to new efficient implementations. We present a generic implementation scheme that, given a linearizable implementation of a sequential specification S, produces an implementation that is locally linearizable wrt S (see Sec. 6). Our implementations show dramatic improvements in performance and scalability. In most cases the locally linearizable implementations scale almost linearly and even outperform state-of-the-art pool implementations. We produced locally linearizable variants of state-of-the-art concurrent queues and stacks, as well as of the relaxed data structures from [20, 25]. The latter are relaxed in two dimensions: they are locally linearizable (the consistency condition is relaxed) and are out-of-order-relaxed (the sequential specification is relaxed). The speedup of the locally linearizable implementation to the fastest linearizable queue (LCRQ) and stack (TS Stack) implementation at 80 threads is 2.77 and 2.64, respectively. Verification of local linearizability, i.e. proving correctness, for each of our new locally linearizable implementations is immediate, given that the starting implementations are linearizable.

2 Semantics of Concurrent Objects

The common approach to define the semantics of an implementation of a concurrent data structure is (1) to specify a set of valid sequential behaviors—the sequential specification, and (2) to relate the admissible concurrent executions to sequential executions specified by the sequential specification—via the consistency condition. That means that an implementation of a concurrent data structure actually corresponds to several sequential data structures, and vice versa, depending on the consistency condition used. A (sequential) data structure D is an object with a set of method calls Σ. We assume that method calls include parameters, i.e., input and output values from a given set of values. The sequential specification S of D

Table 1 The pool axioms (1), (2), (3); the queue order axiom (4); the stack order axiom (5).

(1) $\forall i, j \in \{1, \dots, n\}.\ \mathtt{s} = m_1 \dots m_n \ \wedge\ m_i = m_j \ \Rightarrow\ i = j$
(2) $\forall x \in V.\ \mathtt{r}(x) \in \mathtt{s} \ \Rightarrow\ \mathtt{i}(x) \in \mathtt{s} \wedge \mathtt{i}(x) \prec_{\mathtt{s}} \mathtt{r}(x)$
(3) $\forall e \in \mathtt{Emp}.\ \forall x \in V.\ \mathtt{i}(x) \prec_{\mathtt{s}} \mathtt{r}(e) \Rightarrow \mathtt{r}(x) \prec_{\mathtt{s}} \mathtt{r}(e)$
(4) $\forall x, y \in V.\ \mathtt{i}(x) \prec_{\mathtt{s}} \mathtt{i}(y) \ \wedge\ \mathtt{r}(y) \in \mathtt{s} \ \Rightarrow\ \mathtt{r}(x) \in \mathtt{s} \ \wedge\ \mathtt{r}(x) \prec_{\mathtt{s}} \mathtt{r}(y)$
(5) $\forall x, y \in V.\ \mathtt{i}(x) \prec_{\mathtt{s}} \mathtt{i}(y) \prec_{\mathtt{s}} \mathtt{r}(x) \ \Rightarrow\ \mathtt{r}(y) \in \mathtt{s} \ \wedge\ \mathtt{r}(y) \prec_{\mathtt{s}} \mathtt{r}(x)$

is a prefix-closed subset of Σ^*. The elements of S are called D-valid sequences. For ease of presentation, we assume that each value in a data structure can be inserted and removed at most once. This is without loss of generality, as we may see the set of values as consisting of pairs of elements (core values) and version numbers, i.e. $V = E \times \mathbb{N}$. Note that this is a technical assumption that only makes the presentation and the proofs simpler, it is not needed and not done in locally linearizable implementations. While elements may be inserted and removed multiple times, the version numbers provide uniqueness of values. Our assumption ensures that whenever a sequence $\mathtt{s}$ is part of a sequential specification S, then, each method call in $\mathtt{s}$ appears exactly once. An additional core value, that is not an element, is `empty`. It is returned by remove method calls that do not find an element to return. We denote by `Emp` the set of values that are versions of `empty`, i.e., $\mathtt{Emp} = \{\mathtt{empty}\} \times \mathbb{N}$.

▶ **Definition 1** (Appears-before Order, Appears-in Relation)**.** Given a sequence $\mathtt{s} \in \Sigma^*$ in which each method call appears exactly once, we denote by $\prec_{\mathtt{s}}$ the total *appears-before order* over method calls in $\mathtt{s}$. Given a method call $m \in \Sigma$, we write $m \in \mathtt{s}$ for m *appears in* $\mathtt{s}$. ◇

Throughout the paper, we will use pool, queue, and stack as typical examples of containers. We specify their sequential specifications in an *axiomatic* way [21], i.e., as sets of axioms that exactly define the valid sequences.

▶ **Definition 2** (Pool, Queue, & Stack)**.** A pool, queue, and stack with values in a set V have the sets of methods $\Sigma_P = \{\mathtt{ins}(x), \mathtt{rem}(x) \mid x \in V\} \cup \{\mathtt{rem}(e) \mid e \in \mathtt{Emp}\}$, $\Sigma_Q = \{\mathtt{enq}(x), \mathtt{deq}(x) \mid x \in V\} \cup \{\mathtt{deq}(e) \mid e \in \mathtt{Emp}\}$, and $\Sigma_S = \{\mathtt{push}(x), \mathtt{pop}(x) \mid x \in V\} \cup \{\mathtt{pop}(e) \mid e \in \mathtt{Emp}\}$, respectively. We denote the sequential specification of a pool by S_P, the sequential specification of a queue by S_Q, and the sequential specification of a stack by S_S. A sequence $\mathtt{s} \in \Sigma_P^*$ belongs to S_P iff it satisfies axioms (1) - (3) in Table 1—*the pool axioms*—when instantiating $\mathtt{i}()$ with $\mathtt{ins}()$ and $\mathtt{r}()$ with $\mathtt{rem}()$. We keep axiom (1) for completeness, although it is subsumed by our assumption that each value is inserted and removed at most once. Specification S_Q contains all sequences $\mathtt{s}$ that satisfy the pool axioms and axiom (4)—*the queue order axiom*—after instantiating $\mathtt{i}()$ with $\mathtt{enq}()$ and $\mathtt{r}()$ with $\mathtt{deq}()$. Finally, S_S contains all sequences $\mathtt{s}$ that satisfy the pool axioms and axiom (5)—*the stack order axiom*—after instantiating $\mathtt{i}()$ with $\mathtt{push}()$ and $\mathtt{r}()$ with $\mathtt{pop}()$. ◇

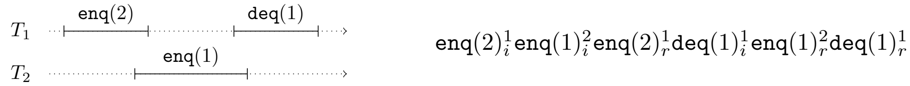
We represent concurrent executions via concurrent histories. An example history is shown in Figure 1. Each thread executes a sequence of method calls from Σ; method calls executed by different threads may overlap (which does not happen in Figure 1). The real-time duration of method calls is irrelevant for the semantics of concurrent objects; all that matters is whether method calls overlap. Given this abstraction, a concurrent history is fully determined by a sequence of invocation and response events of method calls. We distinguish method invocation and response events by augmenting the alphabet. Let $\Sigma_i = \{m_i \mid m \in \Sigma\}$ and $\Sigma_r = \{m_r \mid m \in \Sigma\}$ denote the sets of method-invocation events and method-response events, respectively, for the method calls in Σ. Moreover, let I be the set of thread identifiers. Let $\Sigma_i^I = \{m_i^k \mid m \in \Sigma, k \in I\}$ and $\Sigma_r^I = \{m_r^k \mid m \in \Sigma, k \in I\}$

denote the sets of method-invocation and -response events augmented with identifiers of executing threads. For example, m_i^k is the invocation of method call m by thread k. Before we proceed, we mention a standard notion that we will need in several occasions.

▶ **Definition 3** (Projection). Let $\mathbf{s}$ be a sequence over alphabet Σ and $M \subseteq \Sigma$. By $\mathbf{s}|M$ we denote the projection of $\mathbf{s}$ on the symbols in M, i.e., the sequence obtained from $\mathbf{s}$ by removing all symbols that are not in M. ◇

▶ **Definition 4** (History). A (concurrent) history $\mathbf{h}$ is a sequence in $(\Sigma_i^I \cup \Sigma_r^I)^*$ where
1. no invocation or response event appears more than once, i.e., if $\mathbf{h} = m_1 \dots m_n$ and $m_h = m_*^k(x)$ and $m_j = m_*^l(x)$, for $* \in \{i, r\}$, then $h = j$ and $k = l$, and
2. if a response event m_r^k appears in $\mathbf{h}$, then the corresponding invocation event m_i^k also appears in $\mathbf{h}$ and $m_i \prec_{\mathbf{h}} m_r$. ◇

▶ **Example 5.** A queue history (left) and its formal representation as a sequence (right):



A history is *sequential* if every response event is immediately preceded by its matching invocation event and vice versa. Hence, we may ignore thread identifiers and identify a sequential history with a sequence in Σ^*, e.g., $\texttt{enq}(1)\texttt{enq}(2)\texttt{deq}(2)\texttt{deq}(1)$ identifies the sequential history in Figure 1.

A history $\mathbf{h}$ is *well-formed* if $\mathbf{h}|k$ is sequential for every thread identifier $k \in I$ where $\mathbf{h}|k$ denotes the projection of $\mathbf{h}$ on the set $\{m_i^k \mid m \in \Sigma\} \cup \{m_r^k \mid m \in \Sigma\}$ of events that are local to thread k. From now on we will use the term history for well-formed history. Also, we may omit thread identifiers if they are not essential in a discussion.

A history $\mathbf{h}$ determines a partial order on its set of method calls, the precedence order:

▶ **Definition 6** (Appears-in Relation, Precedence Order). The set of method calls of a history $\mathbf{h}$ is $M(\mathbf{h}) = \{m \mid m_i \in \mathbf{h}\}$. A method call m appears in $\mathbf{h}$, notation $m \in \mathbf{h}$, if $m \in M(\mathbf{h})$. The *precedence order* for $\mathbf{h}$ is the partial order $<_{\mathbf{h}}$ such that, for $m, n \in \mathbf{h}$, we have that $m <_{\mathbf{h}} n$ iff $m_r \prec_{\mathbf{h}} n_i$. By $<_{\mathbf{h}}^k$ we denote $<_{\mathbf{h}|k}$, the subset of the precedence order that relates pairs of method calls of thread k, i.e., the program order of thread k. ◇

We can characterize a sequential history as a history whose precedence order is total. In particular, the precedence order $<_{\mathbf{s}}$ of a sequential history $\mathbf{s}$ coincides with its appears-before order $\prec_{\mathbf{s}}$. The total order for history $\mathbf{s}$ in Fig. 1 is $\texttt{enq}(1) <_{\mathbf{s}} \texttt{enq}(2) <_{\mathbf{s}} \texttt{deq}(2) <_{\mathbf{s}} \texttt{deq}(1)$.

▶ **Definition 7** (Projection to a set of method calls). Let $\mathbf{h}$ be a history, $M \subseteq \Sigma$, $M_i^I = \{m_i^k \mid m \in M, k \in I\}$, and $M_r^I = \{m_r^k \mid m \in M, k \in I\}$. Then, we write $\mathbf{h}|M$ for $\mathbf{h}|(M_i^I \cup M_r^I)$. ◇

Note that $\mathbf{h}|M$ inherits $\mathbf{h}$'s precedence order: $m <_{\mathbf{h}|M} n \;\Leftrightarrow\; m \in M \wedge n \in M \wedge m <_{\mathbf{h}} n$

A history $\mathbf{h}$ is *complete* if the response of every invocation event in $\mathbf{h}$ appears in $\mathbf{h}$. Given a history $\mathbf{h}$, $\texttt{Complete}(\mathbf{h})$ denotes the set of all *completions* of $\mathbf{h}$, i.e., the set of all complete histories that are obtained from $\mathbf{h}$ by appending missing response events and/or removing pending invocation events. Note that $\texttt{Complete}(\mathbf{h}) = \{\mathbf{h}\}$ iff $\mathbf{h}$ is a complete history.

A concurrent data structure D over a set of methods Σ is a (prefix-closed) set of concurrent histories over Σ. A history may involve several concurrent objects. Let O be a set of concurrent objects with individual sets of method calls Σ_q and sequential specifications S_q for each object $q \in O$. A history $\mathbf{h}$ over O is a history over the (disjoint) union of method

calls of all objects in O, i.e., it has a set of method calls $\bigcup_{q \in O}\{q.m \mid m \in \Sigma_q\}$. The added prefix $q.$ ensures that the union is disjoint. The *projection* of $\mathbf{h}$ to an object $q \in O$, denoted by $\mathbf{h}|q$, is the history with a set of method calls Σ_q obtained by removing the prefix $q.$ in every method call in $\mathbf{h}|\{q.m \mid m \in \Sigma_q\}$.

▶ **Definition 8** (Linearizability [23]). A history $\mathbf{h}$ is *linearizable* wrt the sequential specification S if there is a sequential history $\mathbf{s} \in S$ and a completion $\mathbf{h}_c \in \texttt{Complete(h)}$ such that

1. $\mathbf{s}$ is a permutation of $\mathbf{h}_c$, and
2. $\mathbf{s}$ preserves the precedence order of $\mathbf{h}_c$, i.e., if $m <_{\mathbf{h}_c} n$, then $m <_{\mathbf{s}} n$.

We refer to $\mathbf{s}$ as a *linearization* of $\mathbf{h}$. A concurrent data structure D is linearizable wrt S if every history $\mathbf{h}$ of D is linearizable wrt S. A history $\mathbf{h}$ over a set of concurrent objects O is linearizable wrt the sequential specifications S_q for $q \in O$ if there exists a linearization $\mathbf{s}$ of $\mathbf{h}$ such that $\mathbf{s}|q \in S_q$ for each object $q \in O$. ◇

3 Local Linearizability

Local linearizability is applicable to containers whose set of method calls is a disjoint union $\Sigma = \texttt{Ins} \cup \texttt{Rem} \cup \texttt{DOb} \cup \texttt{SOb}$ of insertion method calls `Ins`, removal method calls `Rem`, data-observation method calls `DOb`, and (global) shape-observation method calls `SOb`. Insertions (removals) insert (remove) a *single* value in the data set V or `empty`; data observations return a *single* value in V; shape observations return a value (not necessarily in V) that provides information on the shape of the state, for example, the size of a data structure. Examples of data observations are $\texttt{head}(x)$ (queue), $\texttt{top}(x)$ (stack), and $\texttt{peek}(x)$ (pool). Examples of shape observations are $\texttt{empty}(b)$ that returns true if the data structure is empty and false otherwise, and $\texttt{size}(n)$ that returns the number of elements in the data structure.

Even though we refrain from formal definitions, we want to stress that a valid sequence of a container remains valid after deleting observer method calls:

$$S \mid (\texttt{Ins} \cup \texttt{Rem}) \subseteq S. \tag{1}$$

There are also containers with multiple insert/remove methods, e.g., a double-ended queue (deque) is a container with insert-left, insert-right, remove-left, and remove-right methods, to which local linearizability is also applicable. However, local linearizability requires that each method call is either an insertion, or a removal, or an observation. As a consequence, set is not a container according to our definition, as in a set $\texttt{ins}(x)$ acts as a global observer first, checking whether (some version of) x is already in the set, and if not inserts x. Also hash tables are not containers for a similar reason.

Note that the arity of each method call in a container being one excludes data structures like snapshot objects. It is possible to deal with higher arities in a fairly natural way, however, at the cost of complicated presentation. We chose to present local linearizability on simple containers only. We present the definition of local linearizability without shape observations here and discuss shape observations in [15].

▶ **Definition 9** (In- and out-methods). Let $\mathbf{h}$ be a container history. For each thread T we define two subsets of the methods in $\mathbf{h}$, called in-methods I_T and out-methods O_T of thread T, respectively:

$$\begin{aligned} \mathrm{I}_T &= \{m \mid m \in M(\mathbf{h}|T) \cap \texttt{Ins}\} \\ \mathrm{O}_T &= \{m(a) \in M(\mathbf{h}) \cap \texttt{Rem} \mid \texttt{ins}(a) \in \mathrm{I}_T\} \cup \{m(e) \in M(\mathbf{h}) \cap \texttt{Rem} \mid e \in \texttt{Emp}\} \\ &\quad \cup \{m(a) \in M(\mathbf{h}) \cap \texttt{DOb} \mid \texttt{ins}(a) \in \mathrm{I}_T\}. \end{aligned}$$

◇

Hence, the in-methods for thread T are all insertions performed by T. The out-methods are all removals and data observers that return values inserted by T. Removals that remove the value `empty` are also automatically added to the out-methods of T as any thread (and hence also T) could be the cause of "inserting" `empty`. This way, removals of `empty` serve as means for global synchronization. Without them each thread could perform all its operations locally without ever communicating with the other threads. Note that the out-methods O_T of thread T need not be performed by T, but they return values that are inserted by T.

▶ **Definition 10** (Thread-induced History)**.** Let $\mathbf{h}$ be a history. The thread-induced history $\mathbf{h}_T$ is the projection of $\mathbf{h}$ to the in- and out-methods of thread T, i.e., $\mathbf{h}_T = \mathbf{h}|\,(\mathrm{I}_T \cup \mathrm{O}_T)$. ◇

▶ **Definition 11** (Local Linearizability)**.** A history $\mathbf{h}$ is locally linearizable wrt a sequential specification S if

1. each thread-induced history $\mathbf{h}_T$ is linearizable wrt S, and
2. the thread-induced histories $\mathbf{h}_T$ form a decomposition of $\mathbf{h}$, i.e., $m \in \mathbf{h} \Rightarrow m \in \mathbf{h}_T$ for some thread T.

A data structure D is locally linearizable wrt S if every history $\mathbf{h}$ of D is locally linearizable wrt S. A history $\mathbf{h}$ over a set of concurrent objects O is locally linearizable wrt the sequential specifications S_q for $q \in O$ if each thread-induced history is linearizable over O and the thread-induced histories form a decomposition of $\mathbf{h}$, i.e., $q.m \in \mathbf{h} \Rightarrow q.m \in \mathbf{h}_T$ for some thread T. ◇

Local linearizability is sequentially correct, i.e., a single-threaded (necessarily sequential) history $\mathbf{h}$ is locally linearizable wrt a sequential specification S iff $\mathbf{h} \in S$. Like linearizability [22], local linearizability is compositional. The complete proof of the following theorem and missing or extended proofs of all following properties can be found in [15].

▶ **Theorem 12** (Compositionality)**.** *A history $\mathbf{h}$ over a set of objects O with sequential specifications S_q for $q \in O$ is locally linearizable iff $\mathbf{h}|q$ is locally linearizable wrt S_q for every $q \in O$.*

Proof (Sketch). The property follows from the compositionality of linearizability and the fact that $(\mathbf{h}|q)_T = \mathbf{h}_T|q$ for every thread T and object q. ◀

The Choices Made. Splitting a global history into subhistories and requiring consistency for each of them is central to local linearizability. While this is common in shared-memory consistency conditions [19, 27, 28, 3, 14, 4, 18], our study of local linearizability is a first step in exploring subhistory-based consistency conditions for concurrent objects.

We chose thread-induced subhistories since thread-locality reduces contention in concurrent objects and is known to lead to high performance as confirmed by our experiments. To assign method calls to thread-induced histories, we took a data-centric point of view by (1) associating data values to threads, and (2) gathering all method calls that insert/return a data value into the subhistory of the associated thread (Def. 9). We associate data values to the thread that inserts them. One can think of alternative approaches, for example, associate with a thread the values that it removed. In our view, the advantages of our choice are clear: First, by assigning inserted values to threads, every value in the history is assigned to some thread. In contrast, in the alternative approach, it is not clear where to assign the values that are inserted but not removed. Second, assigning inserted values to the inserting thread enables eager removals and ensures progress in locally linearizable data structures. In the alternative approach, it seems like the semantics of removing `empty` should be local.

An orthogonal issue is to assign values from shape observations to threads. In [15], we discuss two meaningful approaches and show how local linearizability can be extended towards shape and data observations that appear in insertion operations of sets.

Finally, we have to choose a consistency condition required for each of the subhistories. We chose linearizability as it is the best (strong) consistency condition for concurrent objects.

4 Local Linearizability vs. Linearizability

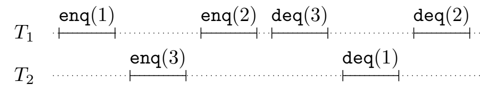
We now investigate the connection between local linearizability and linearizability.

▶ Proposition 1 (Lin 1). In general, linearizability does not imply local linearizability.

Proof. We provide an example of a data structure that is linearizable but not locally linearizable. Consider a sequential specification S_{NearlyQ} which behaves like a queue except when the first two insertions were performed without a removal in between—then the first two elements are removed out of order. Formally, $\mathbf{s} \in S_{\mathsf{NearlyQ}}$ iff

1. $\mathbf{s} = \mathbf{s_1}\mathtt{enq}(a)\mathtt{enq}(b)\mathbf{s_2}\mathtt{deq}(b)\mathbf{s_3}\mathtt{deq}(a)\mathbf{s_4}$ where $\mathbf{s_1}\mathtt{enq}(a)\mathtt{enq}(b)\mathbf{s_2}\mathtt{deq}(a)\mathbf{s_3}\mathtt{deq}(b)\mathbf{s_4} \in S_Q$ and $\mathbf{s_1} \in \{\mathtt{deq}(e) \mid e \in \mathtt{Emp}\}^*$ for some $a, b \in V$, or
2. $\mathbf{s} \in S_Q$ and $\mathbf{s} \neq \mathbf{s_1}\mathtt{enq}(a)\mathtt{enq}(b)\mathbf{s_2}$ for $\mathbf{s_1} \in \{\mathtt{deq}(e) \mid e \in \mathtt{Emp}\}^*$ and $a, b \in V$.

The example below is linearizable wrt S_{NearlyQ}. However, T_1's induced history $\mathtt{enq}(1)\mathtt{enq}(2)\mathtt{deq}(1)\mathtt{deq}(2)$ is not.



◀

The following condition on a data structure specification is sufficient for linearizability to imply local linearizability and is satisfied, e.g., by pool, queue, and stack.

▶ **Definition 13** (Closure under Data-Projection). A seq. specification S over Σ is *closed under data-projection*[1] iff for all $\mathbf{s} \in S$ and all $V' \subseteq V$, $\mathbf{s}|\{m(x) \in \Sigma \mid x \in V' \cup \mathtt{Emp}\} \in S$. ◇

For $\mathbf{s} = \mathtt{enq}(1)\mathtt{enq}(3)\mathtt{enq}(2)\mathtt{deq}(3)\mathtt{deq}(1)\mathtt{deq}(2)$ we have $\mathbf{s} \in S_{\mathsf{NearlyQ}}$, but $\mathbf{s}|\{\mathtt{enq}(x), \mathtt{deq}(x) \mid x \in \{1,2\} \cup \mathtt{Emp}\} \notin S_{\mathsf{NearlyQ}}$, i.e., S_{NearlyQ} is not closed under data-projection.

▶ Proposition 2 (Lin 2). Linearizability implies local linearizability for sequential specifications that are closed under data-projection.

Proof (Sketch). The property follows from Definition 13 and Equation (1). ◀

There exist corner cases where local linearizability coincides with linearizability, e.g., for $S = \emptyset$ or $S = \Sigma^*$, or for single-producer/multiple-consumer histories.

We now turn our attention to pool, queue, and stack.

▶ Proposition 3. The seq. specifications S_P, S_Q, and S_S are closed under data-projection.

Proof (Sketch). Let $\mathbf{s} \in S_P$, $V' \subseteq V$, and let $\mathbf{s}' = \mathbf{s}|(\{\mathtt{ins}(x), \mathtt{rem}(x) \mid x \in V' \cup \mathtt{Emp}\})$. Then, it suffices to check that all axioms for pool (Definition 2 and Table 1) hold for $\mathbf{s}'$. ◀

▶ **Theorem 14** (Pool & Queue & Stack, Lin). *For pool, queue, and stack, local linearizability is (strictly) weaker than linearizability.*

---

[1] The same notion has been used in [7] under the name *closure under projection*.

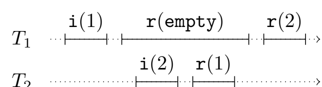


Figure 2 LL, not SC (Pool, Queue, Stack).

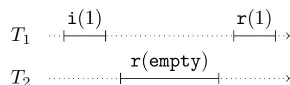


Figure 3 SC, not LL (Pool, Queue, Stack).

Proof. Linearizability implies local linearizability for pool, queue, and stack as a consequence of Proposition 2 and Proposition 3. The history in Figure 2 is locally linearizable but not linearizable wrt pool, queue and stack (after suitable renaming of method calls). ◀

Although local linearizability wrt a pool does not imply linearizability wrt a pool (Theorem 14), it still guarantees several properties that ensure sane behavior as stated next.

▶ Proposition 4 (LocLin Pool). Let $\mathbf{h}$ be a locally linearizable history wrt a pool. Then:

1. No value is duplicated, i.e., every remove method appears in $\mathbf{h}$ at most once.
2. No out-of-thin-air values, i.e., $\forall x \in V.\ \texttt{rem}(x) \in \mathbf{h} \Rightarrow \texttt{ins}(x) \in \mathbf{h} \ \wedge\ \texttt{rem}(x) \not<_{\mathbf{h}} \texttt{ins}(x)$.
3. No value is lost, i.e., $\forall x \in V.\ \forall e \in \texttt{Emp}.\ \texttt{rem}(e) <_{\mathbf{h}} \texttt{rem}(x) \Rightarrow \texttt{ins}(x) \not<_{\mathbf{h}} \texttt{rem}(e)$ and $\forall x \in V.\ \forall e \in \texttt{Emp}.\ \texttt{ins}(x) <_{\mathbf{h}} \texttt{rem}(e) \Rightarrow \texttt{rem}(x) \in \mathbf{h} \ \wedge\ \texttt{rem}(e) \not<_{\mathbf{h}} \texttt{rem}(x)$.

Proof. By direct unfolding of the definitions. ◀

Note that if a history $\mathbf{h}$ is linearizable wrt a pool, then all of the three stated properties hold, as a consequence of linearizability and the definition of S_P.

5 Local Linearizability vs. Other Relaxed Consistency Conditions

We compare local linearizability with other classical consistency conditions to better understand its guarantees and implications.

Sequential Consistency (SC). A history $\mathbf{h}$ is *sequentially consistent* [22, 26] wrt a sequential specification S, if there exists a sequential history $\mathbf{s} \in S$ and a completion $\mathbf{h}_c \in$ `Complete(h)` such that

1. $\mathbf{s}$ is a permutation of $\mathbf{h}_c$, and
2. $\mathbf{s}$ preserves each thread's program order, i.e., if $m <^T_{\mathbf{h}} n$, for some thread T, then $m <_{\mathbf{s}} n$.

We refer to $\mathbf{s}$ as a *sequential witness* of $\mathbf{h}$. A data structure D is sequentially consistent wrt S if every history $\mathbf{h}$ of D is sequentially consistent wrt S.

Sequential consistency is a useful consistency condition for shared memory but it is not really suitable for data structures as it allows for behavior that excludes any coordination between threads [34]: an implementation of a data structure in which every thread uses a dedicated copy of a sequential data structure without any synchronization is sequentially consistent. A sequentially consistent queue might always return `empty` in one (consumer) thread as the point in time of the operation can be moved, e.g., see Figure 3. In a producer-consumer scenario such a queue might end up with some threads not doing any work.

▶ **Theorem 15** (Pool, Queue & Stack, SC). *For pool, queue, and stack, local linearizability is incomparable to sequential consistency.* ◀

Figures 2 and 3 give example histories that show the statement of Theorem 15. In contrast to local linearizability, sequential consistency is not compositional [22].

(Quantitative) Quiescent Consistency (QC & QQC). Like linearizability and sequential consistency, quiescent consistency [11, 22] also requires the existence of a sequential history, a *quiescent witness*, that satisfies the sequential specification. All three consistency conditions impose an order on the method calls of a concurrent history that a witness has to preserve. Quiescent consistency uses the concept of *quiescent states* to relax the requirement of preserving the precedence order imposed by linearizability. A quiescent state is a point in a history at which there are no pending invocation events (all invoked method calls have already responded). In a quiescent witness, a method call m has to appear before a method call n if and only if there is a quiescent state between m and n. Method calls between two consecutive quiescent states can be ordered arbitrarily. *Quantitative quiescent consistency* [24] refines quiescent consistency by bounding the number of reorderings of operations between two quiescent states based on the concurrent behavior between these two states.

The next result about quiescent consistency for pool is needed to establish the connection between quiescent consistency and local linearizability.

▶ Proposition 5. A pool history $\mathbf{h}$ satisfying 1.-3. of Prop. 4 is quiescently consistent. ◀

From Prop. 4 and 5 follows that local linearizability implies quiescent consistency for pool.

▶ **Theorem 16** (Pool, Queue & Stack, QC). *For pool, local linearizability is (strictly) stronger than quiescent consistency. For queue and stack, local linearizability is incomparable to quiescent consistency.* ◀

Local linearizability also does not imply the stronger condition of quantitative quiescent consistency. Like local linearizability, quiescent consistency and quantitative quiescent consistency are compositional [22, 24]. For details, please see [15].

Consistency Conditions for Distributed Shared Memory. There is extensive research on consistency conditions for distributed shared memory [3, 4, 8, 14, 18, 19, 26, 27, 28]. In [15], we compare local linearizability against coherence, PRAM consistency, processor consistency, causal consistency, and local consistency. All these conditions split a history into subhistories and require consistency of the subhistories. For our comparison, we first define a sequential specification S_M for a single memory location. We assume that each memory location is preinitialized with a value $v_{init} \in V$. A read-operation returns the value of the last write-operation that was performed on the memory location or v_{init} if there was no write-operation. We denote write-operations by `ins` and read-operations by `head`. Formally, we define S_M as $S_M = \{\mathtt{head}(v_{init})\}^\star \cdot \{\mathtt{ins}(v)\mathtt{head}(v)^i \mid i \geq 0, v \in V\}^\star$. Note that read-operations are data observations and the same value can be read multiple times. For brevity, we only consider histories that involve a single memory location. In the following, we summarize our comparison. For details, please see [15].

While local linearizability is well-suited for concurrent data structures, this is not necessarily true for the mentioned shared-memory consistency conditions. On the other hand, local linearizability appears to be problematic for shared memory. Consider the locally linearizable history in Figure 4. There, the read values oscillate between different values that were written by different threads. Therefore, local linearizability does not imply any of the

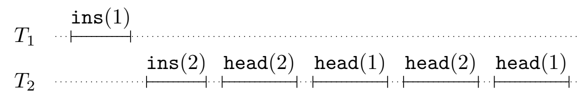


Figure 4 Problematic shared-memory history.

shared-memory consistency conditions. In [15], we further show that local linearizability is incomparable to all considered shared-memory conditions.

6 Locally Linearizable Implementations

In this section, we focus on locally linearizable data structure implementations that are generic as follows: Choose a linearizable implementation of a data structure Φ wrt a sequential specification S_Φ, and we turn it into a (distributed) data structure called LLD Φ that is locally linearizable wrt S_Φ. An LLD implementation takes several copies of Φ (that we call backends) and assigns to each thread T a backend Φ_T. Then, when thread T inserts an element into LLD Φ, the element is inserted into Φ_T, and when an arbitrary thread removes an element from LLD Φ, the element is removed from some Φ_T eagerly, i.e., if no element is found in the attempted backend Φ_T the search for an element continues through all other backends. If no element is found in one round through the backends, then we return `empty`.

▶ Proposition 6 (LLD correctness). Let Φ be a data structure implementation that is linearizable wrt a sequential specification S_Φ. Then LLD Φ is locally linearizable wrt S_Φ.

Proof. Let $\mathbf{h}$ be a history of LLD Φ. The crucial observation is that each thread-induced history $\mathbf{h_T}$ is a backend history of Φ_T and hence linearizable wrt S_Φ. ◀

Any number of copies (backends) is allowed in this generic implementation of LLD Φ. If we take just one copy, we end up with a linearizable implementation. Also, any way of choosing a backend for removals is fine. However, both the number of backends and the backend selection strategy upon removals affect the performance significantly. In our LLD Φ implementations we use one backend per thread, resulting in no contention on insertions, and always attempt a local remove first. If this does not return an element, then we continue a search through all other backends starting from a randomly chosen backend.

LLD Φ is an implementation closely related to Distributed Queues (DQs) [16]. A DQ is a (linearizable) *pool* that is organized as a single segment of length ℓ holding ℓ backends. DQs come in different flavours depending on how insert and remove methods are distributed across the segment when accessing backends. No DQ variant in [16] follows the LLD approach described above. Moreover, while DQ algorithms are implemented for a fixed number of backends, LLD Φ implementations manage a segment of variable size, one backend per (active) thread. Note that the strategy of selecting backends in the LLD Φ implementations is similar to other work in work stealing [30]. However, in contrast to this work our data structures neither duplicate nor lose elements. LLD (stack) implementations have been successfully applied for managing free lists in the fast and scalable memory allocator scalloc [5]. The guarantees provided by local linearizability are not needed for the correctness of scalloc, i.e., the free lists could also use a weak pool (pool without a linearizable emptiness check). However, the LLD stack implementations provide good caching behavior when threads operate on their local stacks whereas a weak pool would potentially negatively impact performance.

We have implemented LLD variants of strict and relaxed queue and stack implementations. None of our implementations involves observation methods, but the LLD algorithm can easily be extended to support observation methods. For details, please see [15]. Finally, let us note that we have also experimented with other locally linearizable implementations that lacked the genericity of the LLD implementations, and whose performance evaluation did not show promising results (see [15]). As shown in Sec. 4, a locally linearizable pool is not a linearizable pool, i.e., it lacks a linearizable emptiness check. Indeed, LLD implementations do not provide a linearizable emptiness check, despite of eager removes. We provide $\mathrm{LL}^{+}\mathrm{D}\ \Phi$, a variant of LLD Φ, that provides a linearizable emptiness check under mild conditions on the starting implementation Φ (see [15] for details).

Experimental Evaluation. All experiments ran on a uniform memory architecture (UMA) machine with four 10-core 2GHz Intel Xeon E7-4850 processors supporting two hardware threads (hyperthreads) per core, 128GB of main memory, and Linux kernel version 3.8.0. We also ran the experiments without hyper-threading resulting in no noticeable difference. The CPU governor has been disabled. All measurements were obtained from the artifact-evaluated Scal benchmarking framework [10, 17, 9], where you can also find the code of all involved data structures. Scal uses preallocated memory (without freeing it) to avoid memory management artifacts. For all measurements we report the arithmetic mean and the 95% confidence interval (sample size=10, corrected sample standard deviation).

In our experiments, we consider the linearizable queues Michael-Scott queue (MS) [29] and LCRQ [31] (improved version [32]), the linearizable stacks Treiber stack (Treiber) [36] and TS stack [12], the k-out-of-order relaxed k-FIFO queue [25] and k-Stack [20] and linearizable well-performing pools based on distributed queues using random balancing [16] (1-RA DQ for queue, and 1-RA DS for stack). For each of these implementations (but the pools) we provide LLD variants (LLD LCRQ, LLD TS stack, LLD k-FIFO, and LLD k-Stack) and, when possible, $\mathrm{LL}^{+}\mathrm{D}$ variants ($\mathrm{LL}^{+}\mathrm{D}$ MS queue and $\mathrm{LL}^{+}\mathrm{D}$ Treiber stack). Making the pools locally linearizable is not promising as they are already distributed. Whenever $\mathrm{LL}^{+}\mathrm{D}$ is achievable for a data structure implementation Φ we present only results for $\mathrm{LL}^{+}\mathrm{D}\ \Phi$ as, in our workloads, LLD Φ and $\mathrm{LL}^{+}\mathrm{D}\ \Phi$ implementations perform with no visible difference.

We evaluate the data structures on a Scal producer-consumer benchmark where each producer and consumer is configured to execute 10^6 operations. To control contention, we add a busy wait of $5\mu s$ between operations. This is important as too high contention results in measuring hardware or operating system (e.g., scheduling) artifacts. The number of threads ranges between 2 and 80 (number of hardware threads) half of which are producers and half consumers. To relate performance and scalability we report the number of data structure operations per second. Data structures that require parameters to be set are configured to allow maximum parallelism for the producer-consumer workload with 80 threads. This results in $k = 80$ for all k-FIFO and k-Stack variants (40 producers and 40 consumers in parallel on a single segment), $p = 80$ for 1-RA-DQ and 1-RA-DS (40 producers and 40 consumers in parallel on different backends). The TS Stack algorithm also needs to be configured with a delay parameter. We use optimal delay ($7\mu s$) for the TS Stack and zero delay for the LLD TS Stack, as delays degrade the performance of the LLD implementation.

Figure 5 shows the results of the producer-consumer benchmarks. Similar to experiments performed elsewhere [12, 20, 25, 31] the well-known algorithms MS and Treiber do not scale for 10 or more threads. The state-of-the-art linearizable queue and stack algorithms LCRQ and TS-interval Stack either perform competitively with their k-out-of-order relaxed counter parts k-FIFO and k-Stack or even outperform and outscale them. For any imple-

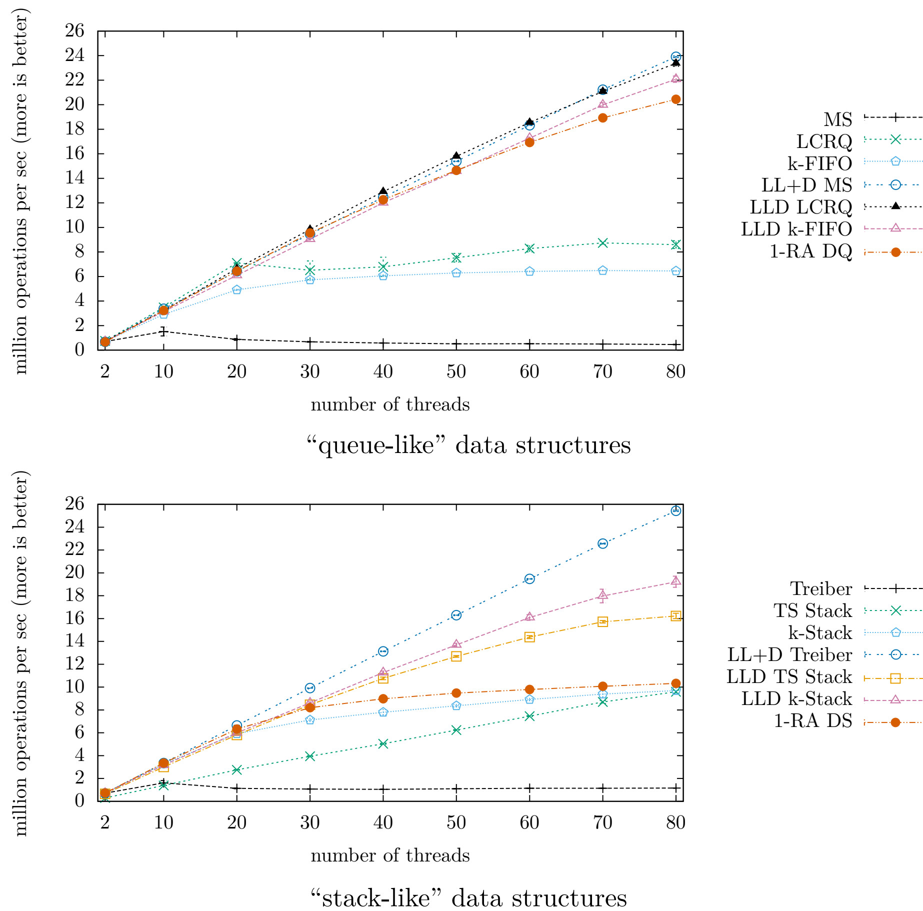


Figure 5 Performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) machine.

mentation Φ, LLD Φ and LL$^+$D Φ (when available) perform and scale significantly better than Φ does, even slightly better than the state-of-the-art pool that we compare to. The best improvement show LLD variants of MS queue and Treiber stack. The speedup of the locally linearizable implementation to the fastest linearizable queue (LCRQ) and stack (TS Stack) implementation at 80 threads is 2.77 and 2.64, respectively. The performance degradation for LCRQ between 30 and 70 threads aligns with the performance of `fetch-and-inc`—the CPU instruction that atomically retrieves and modifies the contents of a memory location—on the benchmarking machine, which is different on the original benchmarking machine [31]. LCRQ uses `fetch-and-inc` as its key atomic instruction.

7 Conclusion & Future Work

Local linearizability splits a history into a set of thread-induced histories and requires consistency of all such. This yields an intuitive consistency condition for concurrent objects that enables new data structure implementations with superior performance and scalability. Local linearizability has desirable properties like compositionality and well-behavedness for container-type data structures. As future work, it is interesting to investigate the guarantees that local linearizability provides to client programs along the line of [13].

References

1. iOS Developer Library, Concurrency Programming Guide, Dispatch Queues. URL: https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html.
2. Y. Afek, G. Korland, and E. Yanovsky. Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In *OPODIS*, pages 395–410, 2010.
3. M. Ahamad, R.A. Bazzi, R. John, P. Kohli, and G. Neiger. The Power of Processor Consistency. In *SPAA*, pages 251–260, 1993.
4. M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, and P.W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
5. M. Aigner, C. M. Kirsch, M. Lippautz, and A. Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *OOPSLA*, pages 451–469, 2015.
6. D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The SprayList: A Scalable Relaxed Priority Queue. In *PPoPP*, pages 11–20, 2015.
7. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On Reducing Linearizability to State Reachability. In *ICALP*, pages 95–107, 2015.
8. S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *POPL*, pages 271–284, 2014.
9. POPL 2015 Artifact Evaluation Committee. POPL 2015 Artifact Evaluation. Accessed on 01/14/2015. URL: http://popl15-aec.cs.umass.edu/home/.
10. Computational Systems Group, University of Salzburg. Scal: High-Performance Multicore-Scalable Computing. URL: http://scal.cs.uni-salzburg.at.
11. J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim. Quiescent Consistency: Defining and Verifying Relaxed Linearizability. In *FM*, pages 200–214, 2014.
12. M. Dodds, A. Haas, and C.M. Kirsch. A Scalable, Correct Time-Stamped Stack. In *POPL*, pages 233–246, 2015.
13. I. Filipovic, P.W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.
14. J.R. Goodman. *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991.
15. A. Haas, T.A. Henzinger, A. Holzer, C.M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, A. Sokolova, and H. Veith. Local Linearizability. *CoRR*, abs/1502.07118, 2016.
16. A. Haas, T.A. Henzinger, C.M. Kirsch, M. Lippautz, H. Payer, A. Sezgin, and A. Sokolova. Distributed Queues in Shared Memory: Multicore Performance and Scalability through Quantitative Relaxation. In *CF*, 2013.
17. A. Haas, T. Hütter, C.M. Kirsch, M. Lippautz, M. Preishuber, and A. Sokolova. Scal: A Benchmarking Suite for Concurrent Data Structures. In *NETYS*, pages 1–14, 2015.
18. A. Heddaya and H. Sinha. Coherence, Non-coherence and Local Consistency in Distributed Shared Memory for Parallel Computing. Technical report, Computer Science Department, Boston University, 1992.
19. J.L. Hennessy and D.A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.
20. T.A. Henzinger, C.M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *POPL*, pages 317–328, 2013.
21. T.A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-Oriented Linearizability Proofs. In *CONCUR*, pages 242–256, 2013.
22. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

23 M. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
24 R. Jagadeesan and J. Riely. Between Linearizability and Quiescent Consistency - Quantitative Quiescent Consistency. In *ICALP*, pages 220–231, 2014.
25 C.M. Kirsch, M. Lippautz, and H. Payer. Fast and Scalable, Lock-free k-FIFO Queues. In *PaCT*, pages 208–223, 2013.
26 L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.
27 R.J. Lipton and J.S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report Nr. 180, Princeton University, Department of Computer Science, 1988.
28 R.J. Lipton and J.S. Sandberg. Oblivious memory computer networking, September 28 1993. CA Patent 1,322,609.
29 M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.
30 M.M. Michael, M.T. Vechev, and V.A. Saraswat. Idempotent Work Stealing. In *PPoPP*, pages 45–54, 2009.
31 A. Morrison and Y. Afek. Fast Concurrent Queues for x86 Processors. In *PPoPP*, pages 103–112, 2013.
32 Multicore Computing Group, Tel Aviv University. Fast Concurrent Queues for x86 Processors. Accessed on 01/28/2015. URL: `http://mcg.cs.tau.ac.il/projects/lcrq/`.
33 H. Rihani, P. Sanders, and R. Dementiev. MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues. *CoRR*, 2014. `arXiv:1411.1209`.
34 A. Sezgin. Sequential Consistency and Concurrent Data Structures. *CoRR*, abs/1506.04910, 2015.
35 N. Shavit. Data Structures in the Multicore Age. *CACM*, 54(3):76–84, March 2011.
36 R.K. Treiber. Systems Programming: Coping with Parallelism. Technical Report RJ-5118, IBM Research Center, 1986.