Toward Linearizability Testing for Multi-Word Persistent Synchronization Primitives

Diego Cepeda
Department of Electrical and Computer Engineering, University of Waterloo, Canada
dcepeda@uwaterloo.ca

Sakib Chowdhury
Department of Electrical and Computer Engineering, University of Waterloo, Canada
sm6chowd@uwaterloo.ca

Nan Li
Department of Electrical and Computer Engineering, University of Waterloo, Canada
n69li@uwaterloo.ca

Raphael Lopez
Department of Mechanical and Mechatronics Engineering, University of Waterloo, Canada
rwlopez@uwaterloo.ca

Xinzhe Wang
Department of Electrical and Computer Engineering, University of Waterloo, Canada
x793wang@uwaterloo.ca

Wojciech Golab
Department of Electrical and Computer Engineering, University of Waterloo, Canada
wgolab@uwaterloo.ca

Abstract

Persistent memory makes it possible to recover in-memory data structures following a failure instead of rebuilding them from state saved in slow secondary storage. Implementing such recoverable data structures correctly is challenging as their underlying algorithms must deal with both parallelism and failures, which makes them especially susceptible to programming errors. Traditional proofs of correctness should therefore be combined with other methods, such as model checking or software testing, to minimize the likelihood of uncaught defects. This research focuses specifically on the algorithmic principles of software testing, particularly linearizability analysis, for multi-word persistent synchronization primitives such as conditional swap operations. We describe an efficient decision procedure for linearizability in this context, and discuss its practical applications in detecting previously-unknown bugs in implementations of multi-word persistent primitives.

2012 ACM Subject Classification Software and its engineering → Synchronization; Theory of computation → Shared memory algorithms; Computer systems organization → Reliability

Keywords and phrases Shared memory, persistent memory, synchronization, multi-word primitives, concurrency, correctness, software testing

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.19

Funding *Diego Cepeda*: Consejo Nacional de Ciencia y Tecnología
Raphael Lopez: University of Waterloo President's Research Award
Xinzhe Wang: University of Waterloo President's Research Award
Wojciech Golab: Natural Sciences and Engineering Research Council (NSERC) of Canada, Discovery Grants Program; Ontario Early Researcher Award; Google Faculty Research Award

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 19; pp. 19:1–19:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Persistent memory makes it possible to recover in-memory data structures following a failure instead of rebuilding them from state saved in slow secondary storage. Implementing such recoverable data structures correctly is challenging as their underlying algorithms must provide both effective concurrency control to harness multi-core parallelism, and recovery procedures to ensure that failures (e.g., process or system crashes) do not corrupt data. The complex states and state transitions of these algorithms make recoverable structures much harder to analyze using traditional means such as rigorous proofs of correctness, particularly with the consideration of failures and subsequent recovery adding a new dimension of difficulty. Automated model checking tools such as PlusCal/TLA+ [23, 24] can be helpful in this context but are difficult to use due to lack of native support for modeling access to persistent memory.

This research aims to augment traditional analysis of correctness with a gray-box software testing approach in which execution histories of synchronization primitives are generated empirically and checked for correctness using a rigorous decision procedure. The correctness criterion under consideration is Herlihy and Wing's widely-adopted *linearizability* property [19], which states that operations applied to an object by a collection of threads must behave as though they take effect instantaneously at some point between their invocation and response events. Deciding linearizability given a history of operations is NP-hard for many data structure types, and often becomes tractable under certain simplifying assumptions [15]. Notably, this holds for histories of primitive read, write, and swap (i.e., Fetch-And-Store) operations if the *reads-from mapping* is known, meaning that the value returned by each operation is either the initial value or the value assigned by a unique write or swap operation, which makes the problem solvable in quasilinear time.

This paper extends and enhances prior work on deciding linearizability for synchronization primitives, such as swaps and atomic counters, as follows:

1. We extend the algorithmic foundations of linearizability testing to multi-word read and conditional swap (i.e., Compare-And-Swap) operations, which are important building blocks of practical data structures for persistent memory.
2. Our techniques take into account operations that are interrupted by failures and do not produce a response, which makes it difficult to deduce their effect.
3. We present an empirical study of a software tool that implements our analysis techniques. The tool is applied to two codebases, and successfully detects previously unknown bugs in both, including one bug that is related directly to the use of persistent memory. We also evaluate the scalability of the tool, and demonstrate that its running time grows nearly linearly with the size of the input history.

2 Model

The model is based closely on Herlihy and Wing's [19]. There are n asynchronous processes, labeled $p_1, \ldots, p_n$, that interact by applying operations on a shared memory with a well-defined initial state. Two types of operations are permitted: an atomic multi-word Read (MwR), and an atomic multi-word Compare-And-Swap (MwCAS) operation. MwCAS is the atomic execution of the pseudocode shown in Figure 1. We assume somewhat unconventionally that the return value of an unsuccessful MwCAS indicates the responsible memory location, which is a crucial piece of information exploited by our gray-box technique (see Section 5.2). Software simulations of multi-word CAS (e.g., [12, 18, 30]) that return a Boolean can be modified easily to meet this specification.

```
Procedure MwCAS( $m_1, \ldots, m_k$: memory locations; $e_1, \ldots, e_k$: expected values;
                 $n_1, \ldots, n_k$; new values)
$old_i := *m_i$ for all $i$, $1 \leq i \leq k$
if $old_i = e_i$ for all $i$, $1 \leq i \leq k$ then
    $*m_i := n_i$ for all $i$, $1 \leq i \leq k$
    // successful CAS
    return 0
else
    // unsuccessful CAS
    return $m_i$ such that $old_i \neq e_i$
```

Figure 1 Sequential specification of multi-word Compare-And-Swap (MwCAS).

Actions of processes are represented using a *history*, which is a sequence of *steps*. Each step is either the invocation of a shared memory operation, or the response of a previously invoked operation. An invocation step is of the form INV(p_i, *op*, *args*) where p_i is a process, *op* is the name of an operation, and *args* are its arguments including the set of memory locations accessed. A response step is of the form RES(p_i, *ret*) where p_i is a process, and *ret* is the return value of the operation. A response step s_r is *matching* with respect to an invocation step s_i in a history H if the following criteria are met: (i) s_r and s_i refer to the same process; (ii) s_i precedes s_r in H; and (iii) no other step by p_i occurs between s_i and s_r in H. Given a history H, its projection onto the steps of a process p_i is denoted by $H|p_i$. A history is *sequential* if it is either empty, or a non-empty sequence of alternating invocation and response steps where each invocation is followed immediately by a matching response. A history H is *well-formed* if, for every process p_i, the projection $H|p_j$ is sequential.

An *operation* in a history H is a pair of steps comprising an invocation and a matching response. Given distinct operations op_1 and op_2, we say that op_1 *happens before* op_2 in H (denoted $op_1 <_H op_2$) if the response step of op_1 precedes the invocation step of op_2 in H. Operations op_1 and op_2 are *concurrent* if neither $op_1 <_H op_2$ nor $op_2 <_H op_1$. A well-formed history H is *linearizable* if there exists a sequential history S, called the *linearization of* H, satisfying the following properties: (i) for each process p_i, $H|p_i = S|p_i$; (ii) for any distinct operations op_1 and op_2 in H, if $op_1 <_H op_2$ then $op_1 <_S op_2$; and (iii) S is *legal*, meaning that operations in S produce responses according to their sequential specification (e.g., Figure 1).

For simplicity of presentation and due to lack of space, we omit from the model the more general definition of linearizability for pending or incomplete operations (i.e., ones lacking responses), and its extensions for persistent memory [2, 7, 17, 21]. In Section 5.3, we will adopt the approach of transforming a given history H to a well-formed history H' such that H has the consistency property under consideration if and only if H' does.

3 Background

The problem of deciding linearizability given a history of operations has been studied widely in the context of atomic read/write registers, and is known to be NP-complete in the general case [15]. It is solvable in polynomial time for reads, writes, and swaps under the assumption that a *reads-from mapping* is known, meaning that each read returns either the initial value at a memory location or the value assigned to this location by a unique write. Such a relation can be established easily for the purpose of software testing by generating execution histories using a driver program that embeds distinct tags (e.g., based on process IDs and per-process counters) in the values written to shared memory.

Assuming that the reads-from relation exists for a history H, our goal is to first decide whether H is linearizable, and if it is not, identify specific anomalies – small subsets of the operations in H that conspire to violate linearizability. Consider the example history illustrated in Figure 2 (a), where a shared register object x is accessed by three writes that assign values 1,2,3, followed by two reads that return 1 and 3. The barbell symbols represent the time intervals of operations, and process IDs are omitted. The history shown is not linearizable because x is overwritten twice between the write and read of value 1, which are denoted by $W(x, 1)$ and $R(x) \to 1$.

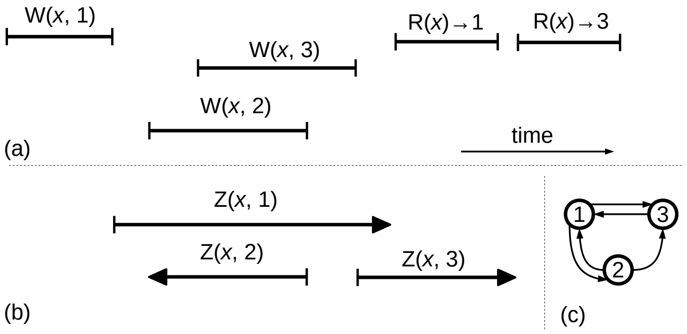

Figure 2 Example execution history (a), its zone-based representation (b), and its graph-theoretic representation (c).

Procedures for deciding linearizability come in two flavours: graph-based, and zone-based. In both cases, the history is first checked for obvious anomalies such as *dangling reads*, which return a value that was never written and is different from the initial value, and *read-write inversions*, where the read of some value v happens before the write of v. Once such cases are ruled out, more subtle anomalies are analyzed. In the graph-based approach inspired by Misra's Axioms for memory access [26], a *precedence graph* $G(V, E)$ is defined whose vertices represent the values read or written, and where an edge (v_1, v_2) exists whenever some operation op_1 that accesses v_1 happens before an operation op_2 that accesses a different value v_2. The input history is linearizable if and only if G has no directed cycles. The graph for our example history is shown in Figure 2 (c), and exhibits several such cycles.

In the zone-based approach of Gibbons and Korach [15], each value v is represented using a time interval called a *zone*, which spans from the time of the earliest response step to the time of the latest invocation step of any operation that reads or writes v. If the earliest response precedes the latest invocation, a *forward zone* occurs, otherwise a *backward zone occurs*. Intuitively, a forward zone for v is a minimal interval of time for which v is continuously the current value, and a backward zone for v is an interval of time containing at least one point at which v is the current value. The history is linearizable if no two forward zones overlap, and no backward zone is contained entirely within a forward zone. The zones for our running example are shown in Figure 2 (b). The overlap between forward zones $Z(x, 1)$ and $Z(x, 3)$, and similarly the containment of backward zone $Z(x, 2)$ within $Z(x, 1)$, indicate that the history is not linearizable.

4 Graph-Theoretic Formulation

Our approach to analyzing the linearizability of persistent synchronization primitives is based on precedence graphs, similarly to the technique described informally in Section 3 based on Misra's Axioms [26]. The key advantage of the graph-based approach is that it represents the constraints on the linearization order in an intuitive way, and (as we show in this paper) can be generalized beyond single-word read/write registers to accommodate a variety of multi-word synchronization primitives. On the other hand, its main weakness is the potentially large number of edges required to construct the graph. For example, given a sequential history with n operations, the precedence graph has $\Theta(n)$ vertices and $\Theta(n^2)$ edges in the worst case since every pair of operations is related by the happens before relation; this can make linearizability analysis quite slow for large execution histories. In comparison, the zone-based algorithm of Gibbons and Korach [15] runs in $O(n \log n)$ time, but is more difficult to analyze and does not generalize easily to multi-word primitives.

Our graph-theoretic approach builds on the simple algorithm described in Section 3, and is inspired by [3, 26]. Given a history H, we construct a *precedence graph* $G(V, E)$ whose vertices represent individual operations rather than the values accessed, and the directed edges represent constraints on the order in which certain pairs of operations must appear in all possible linearizations. A special *genesis vertex* is added to V to represent the initial state of shared memory, and can be regarded as a multi-word write (MwW) operation that creates this state. The precedence graph G is a multigraph, meaning that each pair of vertices can be connected by more than one directed edge. The edge multiset E includes an edge (op_1, op_2) in the following scenarios:

1. *Happens-before edge*: indicates that $op_1 <_H op_2$.
2. *Reads-from edge*: indicates a read-after-write dependency, meaning that op_1 is a multi-word write (genesis vertex) or a successful MwCAS that writes a value v to some memory location m, and op_2 is a multi-word read or successful MwCAS that reads v from m.
3. *Auxiliary edge*: indicates any other precedence constraint on op_1 and op_2.

As explained later on in Section 5, we use two types of auxiliary edges. If op_2 is a successful MwCAS that changes the value at some memory location m from v to v', then the edge (op_1, op_2) indicates a write-after-read dependency where op_1 is a multi-word read that reads v from m. If op_2 is an unsuccessful MwCAS that expects to read a value v at some memory location m and instead encounters a value that is different from v (see Figure 1), then the edge (op_1, op_2) indicates a special type of read-after-write dependency where op_1 is a successful MwCAS that overwrites v at m with a different value.

Our graph-theoretic analysis technique is based on the following assumptions:

▶ **Assumption 1.** *For every history H, for every memory location m, and for every successful MwCAS operation op that writes some value v to m, v is different from the initial value at m and from any value written to m by another successful MwCAS operation.*

▶ **Assumption 2.** *For every history H, for every memory location m, and for every MwCAS operation op that accesses m, the value expected by op at m is a value that was read from m by some MwR or successful MwCAS operation op' such that $op' <_H op$.*

Assumption 1 helps to establish a reads-from mapping, and Assumption 2 simplifies reasoning about unsuccessful MwCAS operations. Both assumptions can be enforced by the tool (e.g., a benchmark program) used to generate histories for testing, without changing the implementation of the synchronization primitive under consideration.

Under the above assumptions, the problem of deciding linearizability given the precedence graph $G(V, E)$ is reduced to the problem of verifying the following structural properties of G:

(i) G is acyclic;

(ii) Every multi-word read or successful MwCAS operation has exactly one incoming reads-from edge for each memory location accessed, and every unsuccessful MwCAS operation has exactly one incoming auxiliary edge; and

(iii) For every memory location m, there exists a directed path of reads-from edges for m that starts at the genesis vertex, and visits every vertex representing a successful MwCAS operation that accesses m. (This path is unique when property i holds.)

Property i is necessary to ensure that all constraints on the linearization order are met, but is not sufficient by itself because operations that return incorrect responses can generate other types of structural anomalies (see Section 6.2). Properties ii–iii compensate for this by ensuring that the state observed by an operation can be attributed to a unique sequence of state transitions starting from the initial state. All three properties can be checked in time linear in the size of G (i.e., $O(|V| + |E|)$), assuming an adjacency list representation where each reads-from edge is labeled with the corresponding memory location m. Property i is established by running depth-first-search (DFS) on the entire graph, and ensuring that no back edges are present. Property ii is established by counting in-edges for the relevant vertices. Property iii is decided using a greedy algorithm that, for each memory location m, repeatedly follows reads-from edges for m starting from the genesis vertex. Supposing that properties i–ii have already been checked, the greedy algorithm either discovers the required path, or else reaches a fork, which proves that such a path does not exist because each vertex under consideration has exactly one incoming reads-from edge for m.

Procedure `ReduceHB` (H: input history)

```
$Edges := \emptyset$
$S :=$ set of operations in $H$
for any operation $op$ in $H$, define Start($op$) and Fin($op$) as the position in $H$ of $op$'s
  invocation and response event, respectively
$L :=$ operations in $S$ sorted in ascending order by Start($op$)
for $op$ in $L$ do
    if $op$ is the last operation in $L$ then
        return $Edges$
    $op' :=$ earliest operation in $L$ such that $op <_H op'$
    $f := \infty$
    $L' :=$ suffix of $L$ from $op'$ (inclusive) onward
    for $op''$ in $L'$ do
        if Start($op''$) $> f$ then
            break from inner for loop
        else
            $Edges := Edges \cup \{(op, op'')\}$
            $f := \min(f, \texttt{Fin}(op''))$
return $Edges$
```

Figure 3 Computation of a minimal subset of happens-before edges.

Before presenting the specifics of read-from and auxiliary edges in Section 5 of the paper, we first describe a general optimization that can reduce the number of edges required to decide linearizability. This optimization is particularly relevant in our work for two reasons. First, our precedence graph uses vertices to represent operations rather than the values accessed by these operations, and tends to generate more edges than the technique presented earlier in Figure 2 (c) because several operations may access one value. Second, because our research emphasizes multi-word primitives, linearizability cannot be checked independently for each memory word, and this puts additional pressure on the algorithm to deal efficiently with large inputs.

The edge set of the precedence graph G tends to be dominated by happens-before edges, whose number in the worst case (i.e., when the input history is nearly sequential) grows quadratically with the number of operations. In contrast, the number of reads-from and auxiliary edges tends to grow linearly under Assumption 1. Our optimization aims to reduce the number of happens-before edges by exploiting the transitivity of the happens-before relation, which makes it possible to derive some precedence constraints indirectly from others. For example, if the input history H contains three operations op_1, op_2, op_3 such that $op_1 <_H op_2$ and $op_2 <_H op_3$, then there is no need to explicitly represent $op_1 <_H op_3$. Applying this observation, it is possible to compute a minimal subset of edges whose transitive closure is the entire happens before relation. The algorithm for selecting such a subset of edges is shown in Figure 3 as procedure `ReduceHB`. Its correctness properties are captured in Theorems 1–2, whose proofs are omitted due to lack of space.

▶ **Theorem 1.** *For any history H, let $G(V,E)$ be the graph whose vertices represent the operations in H, and where E is the set of edges output by* `ReduceHB(`H`)`*. Then for every pair of operations op_1, op_2 in H, $op_1 <_H op_2$ if and only if there is a directed path in G from op_1 to op_2. Furthermore, if H contains three operations op_1, op_2, op_3 such that $op_1 <_H op_2$ and $op_2 <_H op_3$, then E does not contain the edge (op_1, op_3).*

▶ **Theorem 2.** *Let C be the* point contention *for a given history H, which is the maximum number of operations that overlap at a single point in time. Then procedure* `ReduceHB(`H`)` *returns a set of $O(Cn)$ edges where n is the number of operations in H, and it has an implementation with time complexity $O(n \log n + Cn)$.*

Intuitively, procedure `ReduceHB` iterates over all operations in the input history H in increasing order of their start time (outer loop), and for each such operation op it identifies a maximal subset E_{op} of operations that succeed op directly in the partial order $<_H$ (inner loop). It suffices to create happens-before edges from op to each element of E_{op}, as any other operation op' that succeeds op in $<_H$ also succeeds some operation in E_{op}. The size of E_{op} is bounded by the point contention parameter C referred to by Theorem 2 because all pairs of operations in E_{op} are concurrent. For practical purposes, C is a small constant, for example the number of parallel threads used in an experiment to generate an execution history. Supposing that $C \in O(1)$, Theorem 2 implies $O(n \log n)$ running time (same as sorting) and a graph with $O(n)$ happens-before edges. As we will explain later on in Section 5, the precedence graph contains $O(kn)$ reads-from and auxiliary edges for k-word operations, and so our selection of happens-before edges ensures that the entire edge set has size $O(kn)$.

5 Constructing the Precedence Graph

In this section, we describe in more detail the formation of the precedence graph, focusing on the reads-from and auxiliary edges. We begin with a discussion of different variations of the multi-word swap, and assume initially that every operation has both an invocation and matching response. In Subsection 5.3, we finally discuss how to handle operations that are interrupted by failures. We do not discuss (multi-word) write operations, but note that they can be incorporated into our framework fairly easily, and leave the details to future work.

5.1 Multi-Word Reads and Successful MwCAS Operations

For didactic purposes, we first describe the details of dealing with k-word reads and successful swaps only, where $k \geq 1$. Consider an operation $\mathrm{MwR}(m_1 \dots m_k) \to r_1 \dots r_k$ that returns r_i from memory location m_i. The reads-from and auxiliary edges required are derived by generalizing Misra's Axioms [26] to multi-word operations. Specifically, the operation must satisfy the following criteria for each memory location m_i in any linearization L of the given history H, where op_R denotes the above multi-word read:

1. There exists a single operation op (possibly the one represented by the genesis vertex) that writes r_i to m_i, and such that $op <_L op_R$.
2. For any operation op' that writes a value $w_i \neq r_i$ to m_i, either $op' <_L op$ or $op_R <_L op'$.

Criteria 1 and 2 are both needed to ensure that L is legal, particularly that op_R returns a correct value for location m_i. In other words, op_R must return a value that was written to m_i, and moreover this must be the value assigned by the most recent update to m_i that precedes op_R in L. For criterion 1, structural property ii of the graph ensures that op exists, Assumption 1 ensures that op is unique, and the reads-from edge (op, op_R) encodes the constraint $op <_L op_R$. For criterion 2, we add auxiliary edges according to the following procedure: we identify for each memory location m the source vertex op of the incoming reads-from edge for m, and its immediate successor s on the path of reads-from edges referred to by structural property iii. If s exists, we insert an auxiliary edge (op_R, s), which ensures that op' in criterion 2 cannot be linearized between op and op_R.

For a successful MwCAS operation op_C, we have analogous criteria where op_R is replaced with op_C, and criterion 2 is relaxed to handle the case when $op' = op_C$. Reads-from edges are inserted as for MwR operations, but auxiliary edges are not used as otherwise one could form a loop from op_C back to itself. In this case, structural property iii compensates for the lack of auxiliary edges by ensuring that op_C has sufficient outbound reads-from edges.

An example of the dependency graph for a linearizable history is shown in Figure 4. For the sake of clarity, some edges as well as some operations required by Assumption 2 are omitted. This example satisfies structural properties i–iii. A non-linearizable example is then shown in Figure 5, where the swap operation for the state change $B : 1 \to 2$ happens before the read, which makes the response of the read operation stale with respect to memory location B (but not A). In this second example, the swap and the read lie on a cycle, which violates structural property i.

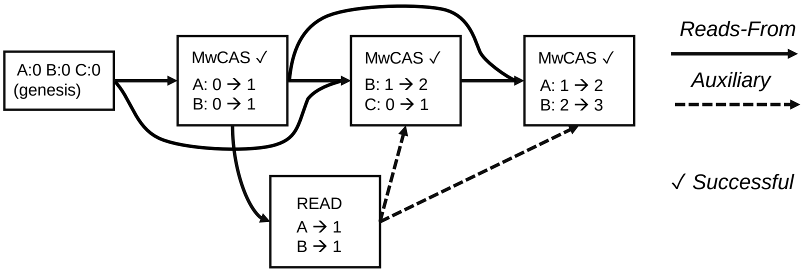

Figure 4 Simplified precedence graph for a linearizable history.

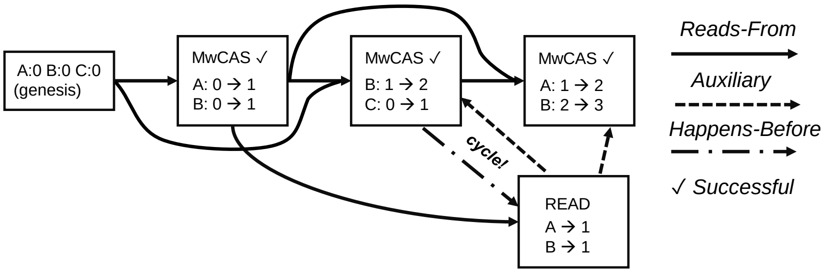

Figure 5 Simplified precedence graph for a non-linearizable history.

5.2 Incorporating Unsuccessful MwCAS Operations

Conditional swap operations present unique challenges that do not exist with unconditional swaps because they can take effect without modifying the state of shared memory. Depending on the implementation, such unsuccessful operations either return the (unexpected) values read, or simply signal that the swap failed. In the former case, the unsuccessful MwCAS is treated like a multi-word read, and so the technique from Subsection 5.1 is sufficient. In the latter case, we assume (as in Figure 1) that the response indicates a memory location m_i for which the observed value was different from the expected value e_i. Also, we require (see Assumption 2) that e_i was read from m_i before the MwCAS. Thus, given some operation op that writes e_i at m_i, the unsuccessful MwCAS$(m_1 \ldots m_k, e_1 \ldots e_k, n_1 \ldots n_k)$ denoted by op_U requires that there exists some operation op' that writes $w_i \neq e_i$ at m_i and satisfies $op <_L op' <_L op_U$ in any linearization L of the given history H. For the purpose of building the dependency graph, we identify op' as the unique successor s to the vertex representing op on the path of reads-from edges for memory location m_i referred to by structural property iii, and insert an auxiliary edge (s, op_U). Intuitively, this encodes the unsuccessful MwCAS reading the value w_i assigned by op', or a newer value.

Figures 6 and 7 illustrate the technique of interpreting an unsuccessful MwCAS in this manner. In Figure 6, the unsuccessful MwCAS takes effect after B is swapped from 1 to 2, and before A is swapped from 1 to 2. Thus, it fails because of B. This is captured by the auxiliary edge from the MwCAS that swapped 2 into B, to the unsuccessful MwCAS. If the unsuccessful MwCAS instead takes effect after A is swapped from 1 to 2 then, as shown in Figure 7, the auxiliary edge closes a cycle, violating structural property i.

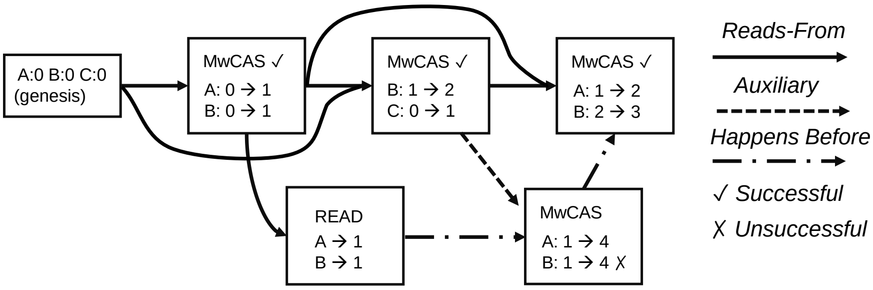

Figure 6 Simplified precedence graph for a linearizable history with an unsuccessful MwCAS that fails because of memory location B.

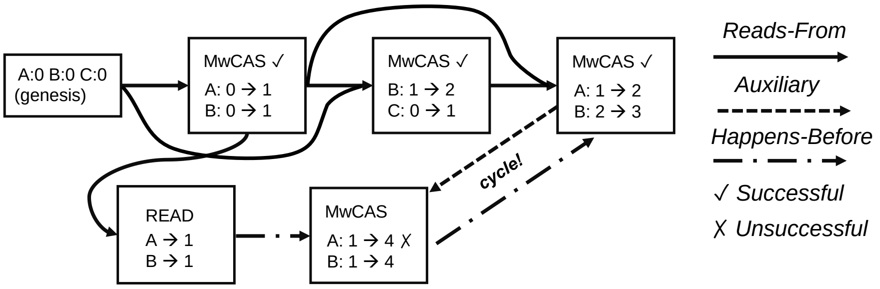

Figure 7 Simplified precedence graph for a non-linearizable history with an unsuccessful MwCAS that fails because of memory location A.

5.3 Operations Interrupted by Crashes

Herlihy and Wing [19] deal with incomplete operations by adding (judiciously chosen) matching responses to a subset of such operations and ignoring the rest. This technicality complicates the analysis of linearizability substantially, and so we have elected to assume in Section 2 that all operations are complete. We now describe how to transform the input history to achieve this property without affecting its linearizability. Regardless of which flavor of linearizability one considers [2, 7, 17, 19, 21], the following rules apply to histories of multi-word reads and MwCAS operations:

1. Incomplete reads can be excluded from the history as they cannot violate linearizability because their responses are not known.
2. An incomplete MwCAS can also be excluded provided that none of the new values it was attempting to swap in has been read by another operation. This holds whether or not the MwCAS actually succeeded and took effect.
3. An incomplete MwCAS whose effect was observed by another operation must be given a matching response. Moreover, the response must indicate that the MwCAS was successful.

The exact placement of the response step stipulated in clause 3 depends on the specific correctness property at hand. For strict linearizability [2], the matching response is placed immediately before the crash. For recoverable linearizability [7], the matching response is inserted at the end of the history, and an auxiliary edge is added from the completed operation to the next operation of the same process, if one exists.[1] For durable linearizability [21], only the matching response is added at the end of the history.

6 Evaluation

This section presents our evaluation of the linearizability analyzer's performance (Subsection 6.1) and effectiveness (Subsection 6.2). All experiments are conducted using a commodity server equipped with four Xeon E5-4620 2.20GHz CPUs. The system has 32 cores total and 256GB of DRAM, 17GB of which are reserved for emulating persistent memory. (The system lacks persistent memory.) The software environment includes Ubuntu Linux 18.04 LTS with kernel version 4.15.0-58-generic, gcc 7.4.0, and OpenJDK 11. Java is used to implement the linearizability analyzer, which comprises 1680 lines of single-threaded code, and can process

[1] Technically speaking, the auxiliary edge should be directed to the next operation applied by the same process on the *same object*. However, given that we are dealing with multi-word operations over a flat address space, we treat the entire collection of memory locations as one shared object.

execution histories obtained by instrumenting persistent synchronization primitives written in any programming language. The C++ compiler is used to compile two implementations of persistent atomic multi-word swap for linearizability analysis.

6.1 Performance Experiments

This section discusses the performance of the analyzer through empirical analysis along two dimensions. First, we assess scalability by measuring the running time of the analyzer on inputs of varying size. Then, we quantify the speed-up due to our optimized method of selecting happens-before edges (Figure 3).

Given the high performance of the synchronization primitives being analyzed, our gray box testing approach has a tendency to generate large execution histories. For example, experiments lasting only one minute can produce histories with millions of operations (n). As a result, it is imperative that running time of the analyzer grows nearly linearly with n, as opposed to a higher-degree polynomial. Naive generation of edges and their selection would instead result in running time and graph size growing quadratically (or worse) with n. As we show through experiment, judicious use of hash tables instead of nested loops, combined with the optimized happens-before edge selection algorithm from Section 4, avoids this problem.

Figure 8 shows the performance of the analyzer with the above optimizations. The analyzer was run repeatedly on increasingly long prefixes of a 1GB execution history log generated using a third-party multi-word Compare-And-Swap implementation (see *CodebaseA* in Section 6.2). Each run was performed 5 times. The average running times and standard deviations are shown in Figure 8. The entire log was deemed linearizable by the analyzer in this case, and same for its prefixes, as expected. Log size is measured in bytes rather than number of operations because both are highly correlated for uniform workloads (approx. 100 bytes per event, two events per operation). Figure 8 shows a definitively linear dependency of both the running time and number of edges on the log size, and hence on n. Minor variations are visible in the running times despite our attempts to control the environment, for example by disabling turbo-boost, which is expected for Java code due to factors such as garbage collection. The trendline still falls within a standard deviation of the runtime for all prefixes.

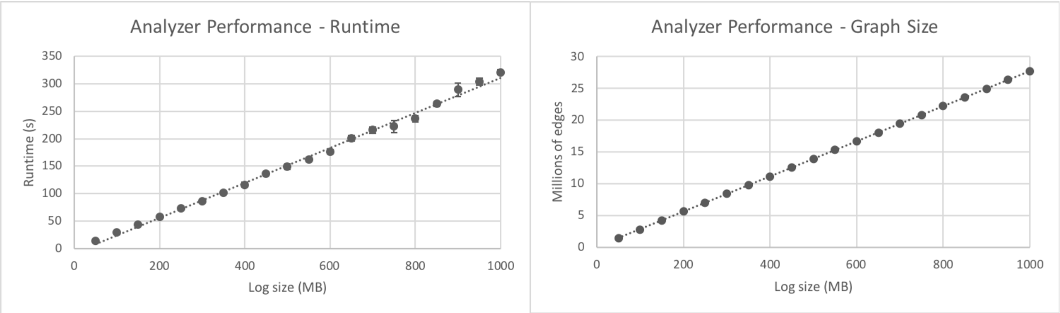

Figure 8 Running time of the analyzer in seconds over prefixes of a 1GB log file (left), and number of edges in the dependency graph constructed by the analyzer (right).

In the next experiment, we compare our optimized implementation of the analyzer against a naive baseline implementation that inserts all possible happens-before edges. Running time is measured over shorter prefixes of the input log used in the scalability experiment presented earlier in Figure 8. The new results are presented in Figure 9, which shows that the `ReduceHB` optimization from Section 4 indeed reduces the total number of edges in the dependency graph from quadratic to linear in the number of operations n. Similarly, the

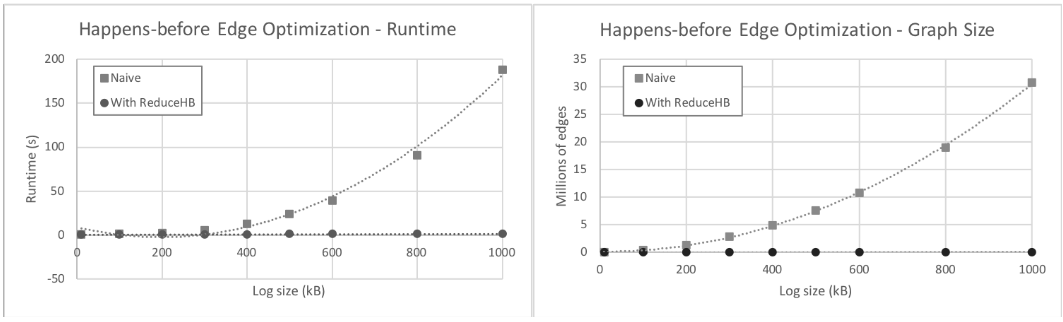

Figure 9 Running time comparison of analyzer with and without the `ReduceHB` optimization from Section 4 (left), and the size of the resulting dependency graphs (right).

running time is reduced to nearly linear. The 1 MB log file, for example, takes 100x longer to process without the optimization. Due to the nonlinearly increasing running time, testing the naive implementation on larger logs quickly became unfeasible.

In a third experiment, we measured the performance penalty due to the logging of execution histories. The results indicate a slowdown of roughly 10%.

6.2 Effectiveness

In this section, we comment on the effectiveness of our linearizability analyzer in the field. To that end, we present a case study in which the analyzer is applied to two implementations of atomic-multi-word swap primitives, in both cases revealing previously unknown bugs.

To test our analyzer, we selected two codebases for analysis: *CodebaseA* is an industrial-grade persistent multi-word compare-and-swap with an open-source implementation [30], and *CodebaseB* is a persistent multi-word unconditional swap developed by a research assistant. We first used a benchmark to generate failure-free single-threaded executions, and discovered no linearizability violations. Next, we considered failure-free concurrent executions with two threads, and detected linearizability violations in *CodebaseB*. As shown in Figure 10, thread t_0 applies an operation that successfully swaps the value i_2 into address A. After that, t_0 performs another swap on the same address A, changing its value successfully from i_2 to i_3. Thread t_1 attempts to swap address A concurrently, but the changes made by t_0 are not reflected in the value read by t_1, which is the previous value i_2 set by t_0 in its first operation. Figure 11 shows a section of the corresponding precedence graph, which violates structural property iii (see Section 4) as no path of reads-from edges for memory location A passes through all the vertices due to a fork.

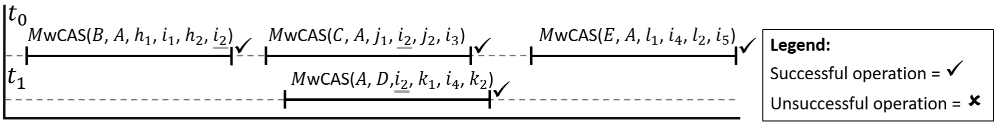
t_0
MwCAS(B, A, h_1, i_1, h_2, i_2) ✓
MwCAS(C, A, j_1, i_2, j_2, i_3) ✓
MwCAS(E, A, l_1, i_4, l_2, i_5) ✓
t_1
MwCAS(A, D, i_2, k_1, i_4, k_2) ✓
Legend:
Successful operation = ✓
Unsuccessful operation = ✗

Figure 10 CodebaseB, non-linearizable execution history indicating a bug.

CodebaseA, showed no linearizability issues in failure-free runs with two threads, but generated non-linearizable histories with three threads. We found spurious events where a previously invoked unsuccessful MwCAS operation interferes with another MwCAS operation that should have succeeded (i.e., causes the latter to become unsuccessful). To further

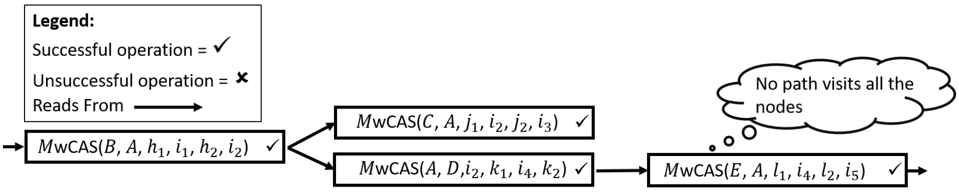
Legend:
Successful operation = ✓
Unsuccessful operation = ✗
Reads From
MwCAS(B, A, h_1, i_1, h_2, i_2) ✓
MwCAS(C, A, j_1, i_2, j_2, i_3) ✓
MwCAS(A, D, i_2, k_1, i_4, k_2) ✓
MwCAS(E, A, l_1, i_4, l_2, i_5) ✓
No path visits all the nodes

Figure 11 CodebaseB, section of precedence graph for the history shown in Figure 10.

understand the spurious events, refer to Figure 12. In this scenario, t_0 begins an operation on addresses B and A, and before t_0 finishes, t_1 begins an operation on addresses C and A. As t_0 finishes the multiword operation, we expect t_1's operation to be unsuccessful due to address A being changed by t_0. Prior to t_0 and t_1 completing their operations, t_2 begins an operation on C and D. Since the completed operation of t_0 does not interfere with the addresses accessed by t_2, and since the operation by t_1 which shares an address in common with t_2 is unsuccessful, we would expect t_2's operation to succeed, but this is not the case. Figure 13 shows a section of the corresponding precedence graph, which violates structural property ii (see Section 4) because the vertex for t_2's unsuccessful MwCAS is disconnected from the rest of the graph.

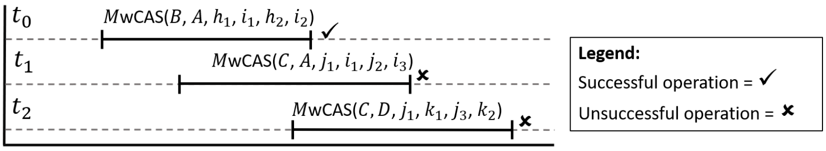
t_0
MwCAS(B, A, h_1, i_1, h_2, i_2) ✓
t_1
MwCAS(C, A, j_1, i_1, j_2, i_3) ✗
t_2
MwCAS(C, D, j_1, k_1, j_3, k_2) ✗
Legend:
Successful operation = ✓
Unsuccessful operation = ✗

Figure 12 CodebaseA, non-linearizable execution history indicating a bug.

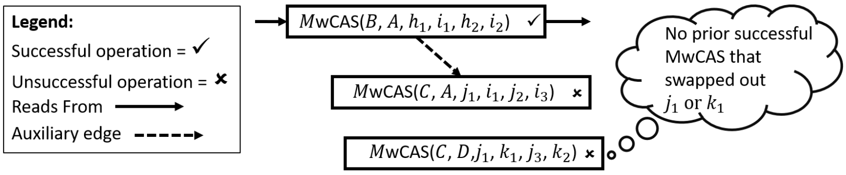
Legend:
Successful operation = ✓
Unsuccessful operation = ✗
Reads From
Auxiliary edge
MwCAS(B, A, h_1, i_1, h_2, i_2) ✓
MwCAS(C, A, j_1, i_1, j_2, i_3) ✗
MwCAS(C, D, j_1, k_1, j_3, k_2) ✗
No prior successful MwCAS that swapped out j_1 or k_1

Figure 13 CodebaseA, section of precedence graph for the history shown in Figure 12.

Before explaining the root cause of the anomalous behavior, we explain briefly the design of *CodebaseA* [30]. MwCAS operations use two types of structures: operation descriptors and word descriptors. Operation descriptors record the arguments, the response of the operation to be performed, the status of the operation (e.g., undecided, successful, unsuccessful), as well as an array of word descriptors. The word descriptors contain the target word address, the expected value to compare against, the new value, and a back pointer to the MwCAS descriptor. The execution of an MwCAS has two phases. *Phase*1 installs a pointer to the MwCAS descriptor in all the target addresses, provided that the current value of a word matches the expected value. The execution path of *Phase*2 depends on the outcome of *Phase*1. For a successful *Phase*1, *Phase*2 installs the new values to the target addresses. For an unsuccessful *Phase*1, *Phase*2 resets any target word that points to the MwCAS descriptor back to its old value. In addition, the algorithm embeds a "dirty bit" in each word to mark data that has been updated but not yet flushed to persistent memory.

The bug in $CodebaseA$ occurs when a thread attempts to install a word descriptor (see line 5 of Algorithm 2 in [30]) using a single-word CAS instruction, and encounters the expected value with the dirty bit set at one of the target memory locations. In Figure 12, such a value is observed by t_2 at location C, which is modified earlier and then restored by t_1. (The bug occurs when t_2 reads C before t_1's unsuccessful MwCAS has a chance to persist the restored value and clear the dirty bit.) The algorithm treats this case as reading a value different from the expected one, and so t_2 completes the MwCAS operation with an unsuccessful status. We modified the code that installs word descriptors to retry if a dirty value is encountered, and this revision eliminated the linearizability anomaly in our tests.

Our case study demonstrates the power of the analyzer to detect previously-unknown bugs on codebases. Once instrumented, we can run and analyze tests with different initial configuration parameters such as number of threads, number of words accessed by each operation, and address space size. The analyzer elucidates subtle issues, reducing the time required to analyze code manually. To conclude this section, we point out that the bug we detected in $CodebaseA$ is related directly to the mechanism used to ensure persistence of the data, namely the dirty bit, and likely would not exist in an atomic multi-word swap implementation designed for conventional volatile memory. Somewhat surprisingly, we were able to catch such a bug without considering crash failures.

7 Related Work

The problems of defining and analyzing linearizability can be traced back to the seminal papers of Lamport and Misra, who formalized the correct behavior of read/write register objects under concurrent access. Lamport [22] introduced safe, regular, and atomic registers in a model where processes may apply read and write operations concurrently, but writes are sequential, and every operation has both an invocation and a response. Misra's Axioms [26] accommodate multi-writer registers, and assume that "all values written by write operations are distinct," which is a natural way to establish a "reads-from" mapping. Herlihy and Wing's linearizability property [19] generalizes Lamport's atomic register in a number of ways: it covers arbitrary typed shared objects; it does not impose restrictions on concurrency (e.g., among writers); and it accommodates pending operations, which lack response events. Linearizability is the gold standard for correctness of shared objects, and is compositional in the following sense: a history involving multiple shared objects is linearizable if and only if all the maximal single-object subhistories are individually linearizable. This property is called *locality*, and can be exploited to parallelize a linearizability analyzer. Horn and Kroening generalized the idea behind locality to operations on the same object, and defined a more fine-grained composition property called *P-compositionality* [20].

Growing interest in implementing shared objects using persistent memory has exposed an important limitation of linearizability: it does not define correct behavior in the case when an operation is interrupted by a failure before it produces a response, and its caller then recovers to perform additional operations on the same object. Several extensions have been proposed to linearizability that address precisely this point, including Aguilera and Frølund's *strict linearizability* [2], Guerraoui and Levy's *persistent atomicity* [17], Berryhill, Golab and Tripunitara's *recoverable linearizability*, Izraelevitz, Mendes and Scott's *durable linearizability* [21], as well as Attiya, Ben-Baruch and Hendler's *nesting-safe recoverable linearizability* [4]. All of these properties are compatible with *detectability* [14], which is the ability to determine the outcome of an operation interrupted by a crash during subsequent recovery.

The problem of deciding whether a history of invocation and response steps satisfies a given consistency property has been studied widely in the context of ordinary linearizability. Gibbons and Korach [15] proved that deciding linearizability is NP-complete for read/write registers. They also introduced the efficient zone-based approach (see Section 4) for the special case when the reads-from mapping is known, including for histories that contain single-word Read-Modify-Write operations (e.g., successful CAS or unconditional swaps) in addition to reads and writes. On the other hand, the graph-based approach is rooted in the theory of database concurrency control [6], and has been used in several studies of consistency in distributed read/write (i.e., key-value) storage systems [3, 5, 16].

Deciding linearizability for types other than read/write registers is a challenging research problem. Automated model checking techniques based on exhaustive state space exploration [1, 9, 10, 25, 28, 29] can accommodate arbitrary data types but are limited to small inputs due to the state space explosion problem. Several more efficient techniques have been devised in the context of collection types (e.g., queue, stack, map, set). Efficient reductions from deciding linearizability to known problems have been proposed by Emmi and Enea [11] (to Horn satisfiability), and by Bouajjani et al. [8] (to control-state reachability). Ozkan, Majumdar and Niksic [27] proved that most histories over collection types can be analyzed efficiently using *hitting families.* Feldman et al. [13] proposed *local view arguments* to simplify linearizability proofs for search trees and skip lists. Our techniques are most similar to [8, 11] in that we decide linearizability automatically in polynomial time, but we focus on a different category of object types and we solve the problem directly rather than by a reduction.

8 Conclusion

This paper described an efficient graph-theoretic technique for deciding linearizability over histories of multi-word read and Compare-And-Swap operations. In our case study, the technique required only small modifications to the synchronization primitive's implementation to capture additional detail regarding unsuccessful Compare-And-Swap operations, and was shown to scale well with the size of the input history. In future work, we plan to extend our results by collecting and analyzing additional histories that include both simulated and real crash failures on a multiprocessor equipped with persistent memory.

References

1 Kiran Adhikari, James Street, Chao Wang, Yang Liu, and Shao Jie Zhang. Verifying a quantitative relaxation of linearizability via refinement. *STTT*, 18(4):393–407, 2016.

2 Marcos Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *HP Labs Tech. Rep. HPL-2003-241*, 2003.

3 Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What Consistency Does Your Key-Value Store Actually Provide? In *Proc. of the Sixth Workshop on Hot Topics in System Dependability (HotDep)*, 2010.

4 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 7–16, 2018.

5 Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. *PVLDB*, 5(8):776–787, 2012.

6 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

7 Ryan Berryhill, Wojciech M. Golab, and Mahesh Tripunitara. Robust Shared Objects for Non-Volatile Main Memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2015.

8 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On Reducing Linearizability to State Reachability. *Inf. Comput.*, 261(Part 2):383–400, 2018.

9 Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In *Proc. of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 330–340, 2010.

10 Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. Model Checking of Linearizability of Concurrent List Implementations. In *Proc. of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 465–479, 2010.

11 Michael Emmi and Constantin Enea. Sound, Complete, and Tractable Linearizability Monitoring for Concurrent Collections. *Proc. ACM Program. Lang.*, 2(POPL):25:1–25:27, 2017.

12 Steven D. Feldman, Pierre LaBorde, and Damian Dechev. A Wait-Free Multi-Word Compare-and-Swap Operation. *International Journal of Parallel Programming*, 43(4):572–596, 2015.

13 Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. Order out of Chaos: Proving Linearizability Using Local Views. In *Proc. of the 32nd International Symposium on Distributed Computing (DISC)*, pages 23:1–23:21, 2018.

14 Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A Persistent Lock-free Queue for Non-volatile Memory. In *Proc. of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, 2018.

15 Phillip B. Gibbons and Ephraim Korach. Testing Shared Memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997.

16 Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. Analyzing consistency properties for fun and profit. In *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 197–206, 2011.

17 Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *Proc. of the 24th International Conference on Distributed Computing Systems (DISC)*, pages 400–407, 2004.

18 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *Proc. of the 16th International Conference on Distributed Computing (DISC)*, pages 265–279, 2002.

19 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

20 Alex Horn and Daniel Kroening. Faster Linearizability Checking via P-Compositionality. In *Proc. of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 50–65, 2015.

21 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proc. of the 30th International Symposium on Distributed Computing (DISC)*, pages 313–327, 2016.

22 Leslie Lamport. On Interprocess Communication, Part I: Basic Formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, 1986.

23 Leslie Lamport. *Specifying systems: the TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

24 Leslie Lamport. The PlusCal Algorithm Language. In *Proc. of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 36–60, 2009.

25 Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.

26 J. Misra. Axioms for Memory Access in Asynchronous Hardware Systems. *ACM Trans. Program. Lang. Syst.*, 8(1):142–153, 1986.

27 Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. Checking Linearizability Using Hitting Families. In *Proc. of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 366–377, 2019.

28 Viktor Vafeiadis. Automatically Proving Linearizability. In *Proc. of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 450–464, 2010.

29 Martin T. Vechev, Eran Yahav, and Greta Yorsh. Experience with Model Checking Linearizability. In *Proc. of the 16th International Workshop on Model Checking Software (SPIN)*, pages 261–278, 2009.

30 Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proc. of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.