DURINN: Adversarial Memory and Thread Interleaving for Detecting Durable Linearizability Bugs

Xinwei Fu, *Virginia Tech;* Dongyoon Lee, *Stony Brook University;*
Changwoo Min, *Virginia Tech*

https://www.usenix.org/conference/osdi22/presentation/fu

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

DURINN: Adversarial Memory and Thread Interleaving for Detecting Durable Linearizability Bugs

Xinwei Fu
Virginia Tech

Dongyoon Lee
Stony Brook University

Changwoo Min
Virginia Tech

Abstract

Non-volatile memory (NVM) has promoted the development of *concurrent crash-consistent* data structures, which serve as the backbone of various in-memory persistent applications. Durable linearizability defines the correct semantics of NVM-backed concurrent crash-consistent data structures, in which linearizability is preserved even in the presence of a crash event. However, designing and implementing a correct durable linearizable data structure remain challenging as developers are to manually control durability (persistence) using low-level cache flush and store fence instructions.

We present DURINN, to the best of our knowledge, the first durable linearizability checker for concurrent NVM data structures. DURINN is based on the novel observation on the gap between linearizability point – when the changes to a concurrent data structure become publicly visible – and durability point – when the changes become persistent. From the detailed gap analysis, we derive three durable linearizability bug patterns that render a linearizable data structure not durable linearizable. To tame the huge NVM states and thread interleaving test space, DURINN statically identifies likely-linearization points and actively constructs adversarial NVM state and thread interleaving settings that increase the likelihood of revealing durable linearizability bugs. DURINN effectively detected 27 (15 new) durable linearizability bugs from 12 concurrent NVM data structures without a test space explosion problem.

1 Introduction

Non-volatile memory (NVM) is becoming widely adopted in various computer systems thanks to its storage-and-memory-like characteristics. Like storage, NVM is persistent across a power cycle and has a high density. Like memory, NVM provides byte-addressability and low-latency properties. A program can persist data in NVM using `load` and `store` instructions without paying storage stack overhead. Notably, Intel's Optane DC Persistent Memory [13, 47] has already been deployed in cloud [3] and supercomputer [2]. ARM also has announced its support for NVM [12, 14]. The upcoming Compute Express Link (CXL) [20] standard introduces cache-coherent CXL-attached NVM card with on-device cache.

The persistence and low-latency properties of NVM have promoted the development of various NVM-backed *concurrent crash-consistent* data structures, which serve as the key enabler of the application-level crash-consistency guarantee. For instance, concurrent NVM hash table is the backbone of NVM-backed `memcached` [7] and `redis` [10]. Concurrent NVM B+tree and hash map are the cores of `pmemkv` [4]. Concurrent NVM B-trees are used in NVM-backed file systems [19, 24–26]. In the event of an application or system crash, or a sudden power failure (a crash hereafter for brevity), an NVM program built on crash-consistent data structures can seamlessly resume its execution from the (recovered) NVM state as if nothing has happened.

Durable linearizability [40] defines the correct semantics of NVM-backed concurrent crash-consistent data structures, as linearizability [33] is the norm correctness standard for traditional (non-NVM) concurrent data structures. At a high level, durable linearizability requires that the effects of completed operations before a crash should remain completed and visible (like linearizability). Additionally, durable linearizability requires that the operation upon a crash be either fully executed ("all" semantic) or not at all executed ("nothing" semantic). However, designing and implementing correct durable linearizable data structures remain very challenging.

The fundamental challenge in developing a durable linearizable NVM data structure lies in *the gap between the linearization point (visibility) and the durability point (persistence).* Concurrent data structures use a synchronization operation (*e.g.*, `compare-and-swap` (`CAS`) in lock-free ones and `lock`/`unlock` in lock-based ones) as *the linearization point* to make one thread's effect visible to other threads. However, the completion of a synchronization operation does not guarantee durability. When the new value of a store (or `CAS`) instruction reaches a cache, it becomes visible, but it is not yet durable until it is written back to the NVM. A data staying in a volatile

cache[1] (as a dirty line) is lost upon a crash, and a cache may evict cache lines in an arbitrary order that is different from the program (store) order. As a result, *the durability point* may come later in time and may appear in a different order with respect to the preceding stores within the same thread or the remote loads from other threads. To ensure durable linearizability, developers have to manually control durability using cache line flush and store fence instructions (*e.g.*, `clwb` and `sfence` in x86 architecture), making durable linearizable NVM programming error-prone.

Unfortunately, existing solutions are not sufficient in testing durable linearizability of concurrent NVM data structures. Prior linearizability testing tools [15, 63] do not consider crash and recovery semantics. NVM-specific crash consistency testing tools either require user-defined custom oracles [22,31,35,43,45,46,51] or are limited to single-threaded NVM programs [30]. It is non-trivial to extend them for durable linearizability because testing space grows exponentially in two dimensions: *crash states* and *thread interleaving*.

This paper presents DURINN, an active and scalable durable linearizability checker for concurrent NVM data structures. DURINN is based on the novel observation on the gap between linearizability point where the changes to a data structure becomes visible and durability point where the changes become persistent and thus remain visible even after a crash. After analyzing what could go wrong if a crash occurs before, after, or between the linearizability and durability points, we derive three durable linearizability (DL) bug patterns that render a linearizable data structure not durable linearizable.

To tame the huge test space, DURINN uses two novel techniques: 1) *adversarial crash state and thread interleaving construction*, and 2) *likely-linearization point inference*. DURINN serves as an adversary of the three DL bug patterns, and actively constructs adversarial crash scenarios that specify which stores to (or not to) persist and which thread interleaving to consider. The intuition behind adversarial crash state construction is to maximize the difference between a constructed crash state and a consistent state preserving DL conditions, thus increasing the likelihood of revealing DL bugs. Furthermore, DURINN employs static program analysis to identify *likely-linearization points* and focuses on testing a program crash before and after those linearization points.

We evaluate DURINN with 13 concurrent NVM data structures, which are highly optimized for NVM and have shown to be more scalable than (simple) NVM hash tables and B-trees used in `memcached`, `redis`, `pmemkv`, etc. DURINN detected 27 (15 new) durable linearizability bugs in 12 data structures. 7 of 15 new bugs have been confirmed by the developers so far. Our evaluation also shows that DURINN can detect concurrent DL bugs (better detection effectiveness) with fewer tests (better scalability), compared to Witcher [30], the state-of-the-art NVM crash-consistency bug detector.

[1] We discuss the implications of (future) persistent cache later in §7.2.

The paper makes the following scientific contributions:

- We present three durable linearizability bug patterns after performing detailed analysis on how a linearizable data structure may violate durable linearizability.
- To our best knowledge, DURINN is the first durable linearizability checker designed for concurrent NVM data structures. The proposed adversarial crash state and thread interleaving construction and likely-linearization point inference allow DURINN to detect DL bugs in an active and scalable manner.
- DURINN reports 27 (15 new) bugs and outperforms the state-of-the-art NVM testing tool in terms of bug detection effectiveness and test space reduction.

2 Background

In this section, we first provide background on linearizability (§2.1) and durable linearizability (§2.2), and then discuss the persistence model used in this paper (§2.3).

2.1 Linearizability

Linearizability [33] is the widely-used correctness standard for concurrent data structures. Formally, linearizability is defined over an existence of an equivalent legal sequential history. Informally, a concurrent data structure is *linearizable* if each operation appears to take effect instantaneously at some moment between the operation begin and end events. If one operation precedes another, then the earlier operation must have taken effect before the next one. If two operations overlap, then their order can be serialized in any arbitrary order. Some pending operations can be thought to be complete.

A *linearization point* (LP) is a program point where an operation takes effect and its effects become visible to other operations. In a lock-based data structure, a critical section (or an `unlock` point) often serves as the linearization point. In a lock-free data structure, the linearization point is typically a single-step atomic instruction (*e.g.*, `CAS`) that makes its change visible to others. We refer to the variable used to make an operation's effect visible as a *synchronization variable* (SV). Atomically updating SV is a linearization point for a writer operation (*e.g.*, `insert`), while reading SV is a linearization point for a reader operation (*e.g.*, `get`).

2.2 Durable Linearizability

Durable linearizability [40] extends linearizability with the notion of a crash. With durable linearizability, a history may include a system-wide crash event (which does not belong to a specific thread) in addition to the operation begin and end events. The definition of precedence order is also extended to incorporate a crash. In durable linearizability, an operation makes its effects visible to others at the linearization point (like linearizability). Additionally, an operation makes its effects persisted at the *durability point* (DP) so that its effects remain completed and visible after a crash.

A *completed operation* refers to an operation whose instructions are all executed, end event is observed, and result

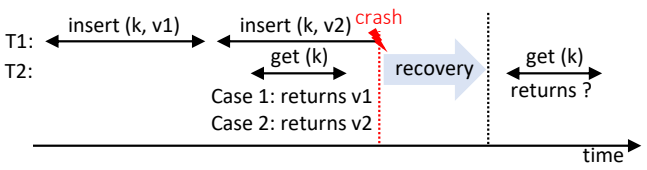


Figure 1: Durable linearizability example. The result of the second `get` (after a crash) depends on the result of the first `get` (before a crash), which also determines if the crashed `insert` takes effect.

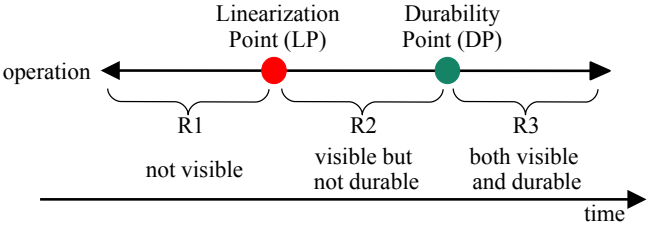


Figure 2: Linearization point and durability point split an operation into three-time intervals as per its visibility and durability guarantees.

is returned. Some crash-consistent programs that can recover from inconsistent NVM state using a custom crash consistency protocol may not require a completed operation to be fully durable.

Durable linearizability requires the following conditions:

- **(C1)** Without a crash, all operations are linearizable.
- **(C2)** If a crash happens, all previously completed operations (before a crash) should remain completed and their effects should remain visible after a crash.
- **(C3)** The operations that have not completed upon a crash could be considered either completed or not. When considered completed, its effect should be visible. In other words, the crashed operation should provide all-or-nothing semantics (fully executed or not at all executed).

Figure 1 illustrates a durable linearizable example. Thread `T1`'s first `insert` completes before a crash, so it should remain visible even after a crash by (C2). If `T2`'s first `get` returns `v1` before a crash, the second `get` after the crash could return either `v1` or `v2`, with a flexibility to follow all or nothing semantics in (C3). However, if `T2`'s first `get` returns `v2` (*i.e.*, it completes) before a crash, then the second `get` after the crash should return `v2`, according to (C1) and (C2).

Prior works such as Line-up [15] and Round-up [63] have demonstrated the practicality of (C1) linearizability testing. DURINN assumes a data structure under test is linearizable without a crash, and focuses on checking if it satisfies the (C2) and (C3) conditions in the presence of a crash.

2.3 Persistence Model

This paper assumes a volatile cache. This is the case for the current Intel x86 architecture with Optane NVM [39, 58] and ARM [12, 14]. The future Compute Express Link (CXL) [20] standard also introduces a cache-coherent interconnect and a CXL-attached NVM card with a volatile on-device cache. We will discuss the implications of persistent cache later in §7.2.

Given a volatile cache, dirty cache lines are lost upon a crash. To control durability, Intel provides cache flush instructions such as `clflush`, `clflushopt`, and `clwb`. When the asynchronous flush `clwb` instruction is used for performance, the store fence `sfence` instruction should be used together to ensure the completion of preceding `clwb` instructions [58]. Similarly, ARM supports `dc cvap` cache flush and `dsb` fence instructions [12, 14]. CXL introduces Global Persistent Flush (GPF) to enforce the persistence ordering on emerging CXL-attached NVM card [20].

3 Durable Linearizability Bugs

This section discusses the gap between linearization point and durability point (§3.1), and presents three durable linearizability bug patterns derived from the gap analysis, along with real-world examples detected by DURINN (§3.2-§3.4).

3.1 The Gap Between LP and DP

As illustrated in Figure 2, the duration of an operation can be partitioned into three regions (`R1`, `R2`, and `R3`) based on the linearization point (LP) and the durability point (DP). At LP, the effect of an operation becomes visible to other threads. At DP, the effect of an operation becomes durable (persisted) so that it can survive a crash and remain visible after a crash.

The bug patterns presented in this section assume that LP is known given an operation. We later in §5.2 discuss the static methods we used to identify *likely-linearization points*. For example, a lock-free `insert()` operation often uses an atomic instruction on a synchronization variable to make its effect visible in a single step. The atomic update (*e.g.*, `CAS`) forms LP, and the following cache line flush and fence instructions (*e.g.*, `clwb` and `sfence`) become DP.

The gap between LP and DP leads to different visibility and durability guarantees. Before LP (region `R1`), the effect of an operation is neither visible nor durable. Between LP and DP (region `R2`), the effect of an operation is visible but not durable. After DP (region `R3`), the effect of an operation is visible as well as durable. Durable linearizability defines different correct/wrong behaviors depending on when a crash occurs: after DP (region `R3`), before DP (regions `R1` and `R2`), and between LP and DP (region `R1`). From the classification, we derive the following three DL bug patterns.

3.2 DL Bug Pattern 1: An Incompletely-Durable Bug

The first *Incompletely-Durable* bug pattern considers a crash *after DP* (in region `R3`). As a crash happens after DP, all the changes made by the crashed operation should be completed and persisted as if no crash has happened. In other words, the crashed operation should provide the "all" (fully-executed) semantic guarantee. After resuming from a crash, if another operation may observe *incompletely durable* effects, then it may produce wrong output violating durable linearizability. Figure 3 illustrates the Incompletely-Durable bug pattern. Since the crash happens after DP of `T1`'s `insert(K,V)`, to be

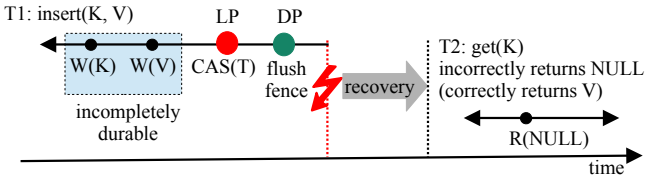


Figure 3: An *Incompletely-Durable* bug pattern. If a crash occurs after DP, the `insert` operation should make all its effects durable completely. Otherwise, the `get` operation after a crash may not be able to see its effect, producing wrong output.

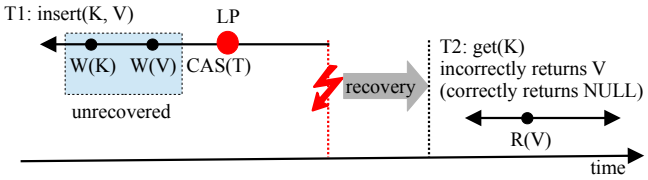


Figure 4: An *Unrecovered-Durable* bug pattern example. If a crash happens before DP, the `insert` operation should recover (undo) any partially durable effect. Otherwise, the `get` operation after a crash may see its partial effect, producing wrong output.

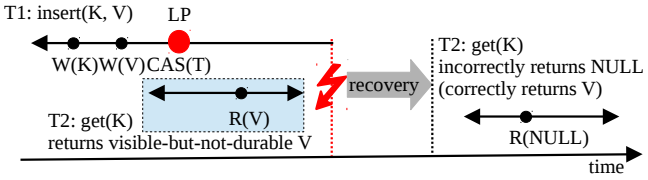
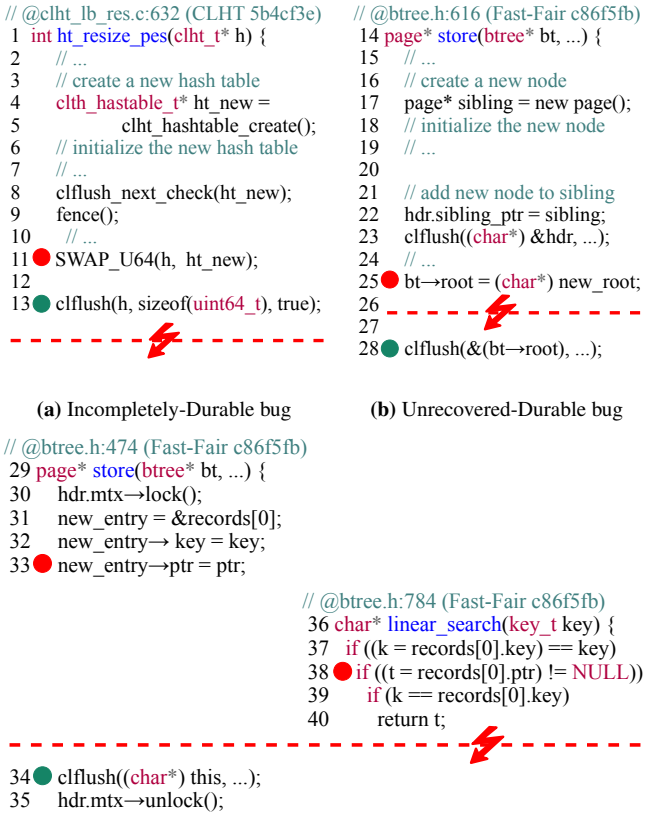


Figure 5: A *Visible-But-Not-Durable* bug pattern. For a crash between LP and DP of `insert`, the concurrent `get` may observe and return the visible-but-not-durable value. However, the second `get` after a crash may not be able to return the same value as the effect of `insert` is not durable.



(a) Incompletely-Durable bug

(b) Unrecovered-Durable bug

(c) Visible-But-Not-Durable bug

Figure 6: Durable linearizability bug examples in P-CLHT [44] (a) and Fast-Fair [34] (b) and (c). A red circle represents LP; green represents DP; and a red lightning bolt represents a crash.

durable linearizable, `T2`'s `get(K)` should return `V` after the recovery.

To avoid *Incompletely-Durable* bugs, all the preceding stores must be persisted before the store (or `CAS`) on a synchronization variable becomes persisted (DP), using cache line flush and fence instructions. This is analogous to the linearizability programming idiom in which all the preceding stores must be visible before the store (or `CAS`) on a synchronization variable become visible (LP), using a fence instruction.

Figure 6(a) shows a part of rehashing code in P-CLHT [44], a concurrent NVM hash table. The code first allocates a new hash table (line 4), updates/persists the new hash table (lines 6-7), and then atomically sets the root pointer `h` to the new hash table, making its effect visible (line 11, which is LP). However, `clflush_next_check` at line 8 does not flush all the updated NVM data in the new hash table and leaves a part unpersisted (an Incompletely-Durable bug). If a crash happens after DP – after persisting the root pointer `h` (line 13), the inserted key-value data after a crash may be lost, violating durable linearizability.

3.3 DL Bug Pattern 2: An Unrecovered-Durable Bug

The second *Unrecovered-Durable* bug pattern considers a crash *before DP* (in regions `R1` and `R2`). As a crash happens before DP, any temporal change made by the crashed operation should not be visible after the resumption. That is, the crashed operation should support the "nothing" (not-at-all-executed) semantic. After resuming from a crash, if another operation may observe *unrecovered durable* effects, it may produce wrong output violating durable linearizability. Figure 4 illustrates the Unrecovered-Durable bug pattern. Since the crash happens before DP of `T1`'s `insert(K,V)`, to be durable linearizable, `T2`'s `get(K)` should not return `V` after the recovery.

To avoid *Unrecovered-Durable* bugs, a durable linearizable data structure may opt to buffer/undo the effects of preceding stores before DP, or embed a custom logic to safely ignore partial NVM updates: *e.g.*, read key `K` and value `V` only if token `T` is set. This pattern is called "guarded protection" [30] and we discuss it in detail in §5.2.

Figure 6(b) shows an *Unrecovered-Durable* bug from Fast-Fair [34], a lock-based NVM B+tree. While splitting a node, it first creates a new node (line 17) and initializes the new node

(lines 18-19). Then it adds the new node to the sibling of the current node (line 22) and persists the change (line 23). Later, it sets the new root (line 25, which is LP). The `new_root` in line 25 is the new root node of the B+tree after a node split. The node, which the `hdr` belongs to, is a child of the new root. If a crash happens before persisting the new root node (line 28, DP), the B+tree will be in an illegal state in which the root node has a sibling node. Any further operation leads to a program crash and will lose all previously completed operations, violating durable linearizability.

3.4 DL Bug Pattern 3: A Visible-But-Not-Durable Bug

The last *Visible-But-Not-Durable* bug pattern considers a crash *between LP and DP* (in region `R2`). If a crash happens between them, the effect of the current operation may be *visible but not yet durable*. As it is visible, another concurrent operation can see the effect and take an action based on the observation: *e.g.*, returning a non-durable value.

Figure 5 illustrates an example. While thread `T1` is performing an `insert(K,V)` operation and just finishes executing its LP but not DP, thread `T2` performs a concurrent `get(K)` operation. The concurrent `get(K)` sees the non-durable effect of `insert(K,V)` and returns the value `V`. As the `get(K)` is completed before a crash, to be durable linearizable, `T2`'s second `get(K)` after the recovery should return `V` as well, but it cannot as the effect of `insert(K,V)` has not been persisted. Note that Visible-But-Not-Durable bugs may also occur between concurrent writers, say two `insert(A)` and `insert(B)` operations in a sorted linked list. The later `insert(B)` operation in the linearizable order may see the effect of the earlier `insert(A)` operation, adding `B` after `A`. Durable linearizability may be violated if a crash occurs after `insert(B)` completes but before `insert(A)` finishes.

To avoid *Visible-But-Not-Durable* bugs, an operation (later in the linearizable order) may be designed to wait until the earlier operation passes its DP. Alternatively, a lock-free design may use a "persistence-helping" mechanism [21, 28]. Suppose operation `A` updated `X` but did not persist it yet. Another concurrent operation `B` wants to take actions (*e.g.*, takes a different branch, persists other data) based on the value of `X`. Then `B` "helps" persist `X` on behalf of `A`. If a lock-free data structure does not implement a similar helping mechanism correctly and if `B` relies on unpersisted updates from `A`, then a Visible-But-Not-Durable bug may happen. The helping logic is analogous to the linearizability programming idiom in which one thread helps fix temporal inconsistency on behalf of another thread.

Figure 6(c) shows a *Visible-But-Not-Durable* bug from Fast-Fair [34]. The left code (`store`) and the right code (`linear_search`) are parts of `insert` and `get` operations, respectively. An `insert` operation first acquires the lock (line 30) then writes `key` and `ptr` (lines 31-33). It then persists the writes (line 34) and releases the lock at the end (line 35). Since Fast-Fair allows concurrent (non-blocking) `get` operations while splitting a node, linking a new node is LP for `insert` (line 33). On the other hand, the `get` operation refers to `ptr` (line 38, which is LP for `get`) while checking if there is any key change in-between by reading it twice (lines 37, 39). Suppose `linear_search` is scheduled between the LP (line 33) and DP (line 34) of `store` as shown in the figure. The concurrent `get` operation can read visible-but-not-durable data. If the crash happens before `insert`'s DP (line 34). After the recovery, the previously returned data cannot be accessed anymore because unpersisted data will be lost upon a crash. Thus, Fast-Fair violates durable linearizability.

4 Overview of Our Approach

In this section, we will first discuss the huge testing space as the main challenge in detecting durable linearizability bugs (§4.1). We then provide an overview of our two major techniques – (1) adversarial NVM state and thread interleaving construction (§4.2) and (2) likely-linearization point inference (§4.3) – designed to address the test space challenge.

4.1 Challenges in Detecting DL Bugs

Existing solutions are not sufficient in testing durable linearizability of concurrent NVM data structures. Traditional linearizability testing tools, such as Line-up [15] and Round-up [63], do not consider crash and recovery semantics. Most NVM-specific crash consistency bug detection tools (*e.g.*, Yat [43], PMTest [46], XFDetector [45], Agamotto [51], Jaaru [31], and PMDebugger [22]) are not designed for durable linearizability, and instead require user-defined custom oracles or consistency checkers. Some (*e.g.*, Witcher [30]) are limited to testing single-threaded NVM programs. We discuss related work in detail in §9.

It is non-trivial to extend existing NVM testing tools such as Yat and Witcher for durable linearizability because testing space grows exponentially in two dimensions: *NVM crash states* and *thread interleaving*. The crash state test space is huge since a crash can happen any time during an execution and a volatile cache can evict cache lines in an arbitrary order. For example, Yat [43], an exhaustive crash consistency testing tool, attempts to test 10^{31} crash states for an NVM hash table with 2000 operations [30]. Moreover, the number of thread interleaving grows exponentially (n^k) with the number of threads (n) and the number of steps (k) in each thread.

4.2 Adversarial NVM State and Thread Interleaving

We propose an adversarial technique to effectively explore the huge testing space in finding durable linearizability bugs. Instead of exhaustively or randomly exploring the testing space, we actively construct adversarial NVM states and adversarial thread interleavings, which are likely to trigger the three DL bug patterns discussed in §3. To the best of our knowledge, DURINN is the first work using an adversarial testing approach for bug detection in NVM programs.

Adversarial NVM state construction. For each DL bug pattern and a given crash location (*e.g.*, before or after DP),

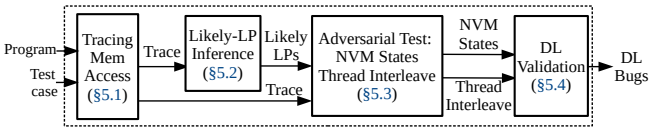


Figure 7: The overall architecture of DURINN.

DURINN determines which preceding stores should be or should not be persisted to increase the likelihood of triggering the DL bugs. For example, when testing an Incompletely-Durable bug after DP, DURINN adversarially constructs the (worst-case) NVM state where an update on a synchronization variable is persisted (at DP) but the preceding stores are not as persisted as possible. This way, DURINN can maximize the incompleteness of durability of a target operation, increasing the chance to break its "all" (fully-executed) semantic guarantee. Note that DURINN constructs only feasible NVM states while obeying the persistence model of a processor and program order semantics (*e.g.*, TSO for x86).

Adversarial thread interleaving construction. DURINN constructs adversarial thread interleaving only when thread interleaving is indispensable to trigger the DL bug patterns. The Incompletely-Durable and Unrecovered-Durable bugs do not require concurrent operations to trigger, so DURINN tests those bug patterns in a single-threaded mode. On the other hand, Visible-But-Not-Durable requires concurrent operations. The main challenge in triggering the Visible-But-Not-Durable bug is that two (or more) concurrent operations must be precisely scheduled in a very narrow window between LP and DP. DURINN adversarially constructs a thread interleaving such that a concurrent operation is scheduled between LP and DP of another operation.

4.3 Likely-Linearization Point Inference

Our adversarial NVM state and thread interleaving construction requires the knowledge of LP locations. Manual annotation of LPs would be error-prone and it makes DURINN not automatic. A naive approach, considering all stores as LPs, would lead to too many tests.

To address the problem, DURINN infers likely-LPs from source code based on the common concurrent NVM programming practices: (1) atomic instructions are used in concurrent programs for synchronization; (2) concurrent programs usually make a memory region visible to other threads after initialing the memory region; (3) NVM programs usually use guarded-protection [30] to ensure persistence atomicity. DURINN employs static program analysis to identify the above programming practices and infer likely-LPs. They are then fed to our adversarial NVM state and thread interleaving construction. The inferred likely-LPs are not necessary to be precise. A false positive LP will only lead to more tests. As far as we know, DURINN is the first work that statically infers linearization points from concurrent NVM programs.

```
// writer
1  *key_ptr = key;
2  *val_ptr  = val;
3  flush(key_ptr);
4  flush(val_ptr);
5  fence();
6
7  flag = 1; // set guardian
8  flush(&flag);
9  fence();

// reader
10  // guardian read
11  if ( flag == 1)
12     func(key_ptr, val_ptr);
```

(a) Guarded-Protection

```
1  // Memory allocation
2  Node* new_node = alloc(sz);
3  // Initialization
4  new_node→key = key;
5  new_node→val =  val;
6  new_node→next = NULL;
7  flush(&new_node→key);
8  flush(&new_node→val);
9  flush(&new_node→next);
10 fence();
11
12 // Add node to the core
13 core→tail = new_node;
14 flush(&core→tail);
15 fence();
```

(b) Publish-after-Initialization

Figure 8: Examples for *Guarded-Protection* and *Publish-after-Initialization* from (a) CCEH [50] and (b) NVTraverse [28]. Likely-linearization points are at line 7 and 13 in (a) and (b), respectively.

5 Design of DURINN

We present the overall architecture of DURINN at Figure 7. DURINN takes as input a target NVM data structure and a test case (a sequence of operations, such as `insert`, `delete`, and `get`) and reports detected durable linearizability bugs. DURINN first instruments a program and runs a test case to collect a memory trace (§5.1). DURINN then infers likely-linearization points from the trace (§5.2). Given the memory trace and the identified likely-linearization points, DURINN performs adversarial NVM state and thread interleaving construction (§5.3) to generate a collection of crashed NVM images and thread schedules to test. Lastly, DURINN validates the generated crashed NVM images along with the generated thread schedules to detect durable linearizability bugs (§5.4).

5.1 Tracing Memory Accesses

DURINN instruments all NVM memory accesses (load, store[2]) and NVM heap allocation. DURINN also traces control flow transfers (branch, function call) because our likely-linearization point inference (§5.2) relies on program dependence analysis. To track the persistent state (*i.e.*, whether an NVM address is persisted or not) for adversarial NVM state construction (§5.3), we instrument all flush and memory fence instructions. We also trace lock operations for adversarial thread interleaving construction (§5.3). For durable linearization validation (§5.4), we trace the value of each store instruction.

We implement an LLVM compiler pass [11] for the instrumentation and execute the instrumented binary with a test case to collect an execution trace. To ensure the total ordering in the execution trace for the analysis of multi-threaded programs, we protect our tracing code using a global mutex.

5.2 Likely-Linearization Point Inference

DURINN infers likely-linearization points by analyzing three concurrent NVM programming practices: *Atomic instruction*, *Guarded-Protection* and *Publish-after-Initialization*.

Atomic Instruction. In lock-free data structures, atomic instructions are typically used to update a synchronization variable and to make the effect of an operation atomically visible to other threads. Thus, DURINN identifies atomic instructions (*e.g.*, `CAS`, `fetch-and-add`) as likely-linearization points for lock-free writer operations (*e.g.*, `insert`).

Guarded-Protection. Guarded protection is a widely used NVM programming pattern (*e.g.*, key-value store, persistent data structure, file systems) to ensure atomic persistence of data [30]. A flag variable called *"guardian"* denotes whether the *"guarded data"* is valid or not. Thus, writing or reading a guardian is a linearization point. In Figure 8(a), for instance, a writer ensures that key and value are persisted before the `flag`, a guardian, is persisted. Also, a reader check if the `flag` (line 11) is set before reading the key and value (*"guarded read"*). Writing to the `flag` (line 7) is writer's linearization point since the changes become visible after setting the `flag`.

Based on this observation, DURINN performs program analysis to identify any stores to guardians. DURINN first finds out the guarded read pattern in the code to identify guardian candidates from conditional branch instructions. From the branch condition variables, DURINN performs the backward dataflow analysis to identify NVM memory addresses that are data-dependent on the branch condition variables. Then DURINN marks the stores to those NVM memory addresses as likely-linearization points.

Publish-after-Initialization. As an optimization to reduce persistence overhead, many NVM program follows so-called publish-after-initialization steps when adding a new memory object in the global data structure: (1) first allocating an NVM memory, (2) initializing the memory, and finally (3) linking (publishing) the memory to the global structure. For example, in Figure 8(b), a node is allocated first (line 2), then initialized (lines 4-10), finally is linked to the global list (`core→tail` at lines 13-15). The benefit of the publish-after-initialization idiom is that any writes to the new memory (lines 4-10) are not externally visible so that the persistence ordering of the writes in the initialization phase is relaxed until the new memory is published (line 13), improving performance (only one `fence` is needed at line 10).

Based on this NVM programming idiom, we filter out all the stores to newly allocated memory regions within an operation, and exclude them from likely-linearization points. We found that this pruning is highly useful for operations requiring many writes, such as node split/merge operations for a tree and a rehashing operation for a hash table.

[2]Non-temporal stores are supported/modeled as store+flush.

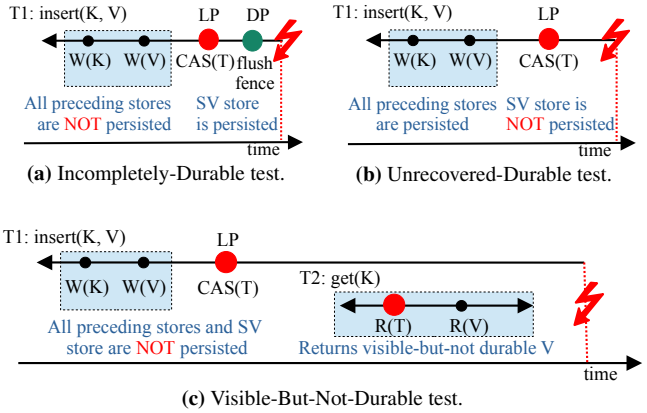


Figure 9: Adversarial test strategies for Incompletely-Durable, Unrecovered-Durable, and Visible-But-Not-Durable bugs. LP: linearization point. DP: durability point. SV: synchronization variable.

5.3 Adversarial NVM State and Thread Interleaving Construction

In this section, we first describe our adversarial construction approaches for each DL bug pattern (§5.3.1, §5.3.2, and §5.3.3). We then introduce our cache/NVM simulations to generate feasible NVM states (§5.3.4) for the validation.

5.3.1 Incompletely-Durable Bug Pattern

Testing Incompletely-Durable bugs can be performed for each operation in isolation without considering concurrent operations. When a crash happens after DP, to be durable linearizable, the crashing operation should provide the "all" (fully-executed) semantic and ensure that its effect remains visible after a crash (Figure 3). Then, the adversarial NVM state that increases the chance to trigger Incompletely-Durable bugs for a crash after DP would be to make all the preceding stores as unpersisted as possible. In other words, we artificially attempt to create a feasible yet worst NVM state that many updates made by an operation are not persisted.

Figure 9a illustrates our adversarial NVM state construction for `insert(K,V)` in which an atomic update to a synchronization variable `T` serves as LP and persisting it serves as DP. The adversarial NVM state would be to make the change to `T` persisted, but leave the changes to key and value unpersisted so that the new key and value data is not visible after a crash even though the synchronization variable `T` says differently. Note that we attempt to leave stores unpersisted only if possible. We do not force. We obey memory consistency and persistence model (*e.g.*, the semantics of fence, flush).

5.3.2 Unrecovered-Durable Bug Pattern

Testing Unrecovered-Durable bugs can also be performed for each operation in isolation. If a crash happens before DP, for durable linearizability, the crashing operation should provide the "nothing" (not-at-all-executed) semantic and ensure that any partial update is not visible after a crash (Figure 4). Then,

the adversarial NVM state that stress-tests the data structure under test to expose Unrecovered-Durable bugs for a crash before DP would be to make all the preceding stores as persisted as possible. That is, we are interested in constructing a feasible yet worst NVM state that many updates made by an operation are persisted, stress-testing its recovery logic.

Figure 9b shows our adversarial NVM state construction for the same `insert(K,V)` example in which `CAS(T)` is LP and persisting it is DP. We construct the adversarial NVM state such that the changes to key and value are persisted, but not the synchronization variable `T`. This way, the new key and value data may be visible after a crash when the synchronization variable `T` says they should not.

5.3.3 Visible-But-Not-Durable Bug Pattern

The Visible-But-Not-Durable bugs are related to the case where an operation takes an action after observing a visible-yet-not-durable state of another concurrent operation (Figure 5). Unlike the prior two bug patterns, testing Visible-But-Not-Durable bugs should be performed in a context sensitive manner. Figure 9c illustrates our adversarial NVM state and thread interleaving method for Visible-But-Not-Durable bugs, which requires the following three conditions.

Requirements. First, DURINN needs *(1) racy operations*. In Figure 9c, thread T1's `insert(K,V)` writes (`CAS`) on `T` and T2's `get(K)` reads `T`. Second, DURINN needs some *(2) prefix operations* (a sequence of other operations to execute before testing racy operations) that construct the preconditions for a race condition to be triggered. For example, an NVM data structure should be in a certain state (*e.g.*, initiating a resizing or node splitting process) to exhibit a race condition. Last, DURINN needs to control *(3) precise thread interleaving* in which a thread makes a progress based on another thread's visible-but-not-durable effect and a crash happens between LP and DP as illustrated in Figure 9c.

Challenges. However, constructing the test scenarios that satisfy all the three conditions is very challenging because not only search space is huge but also the three conditions are inter-dependent. For example, two racy operations with one sequence of prefix operations may not be racy any more with another sequence of prefix operations.

Our Approach. We propose techniques to find out adversarial (1) racy operations, (2) prefix operations, and (3) thread interleaving in a scalable manner by analyzing a *single-threaded* execution trace. Figure 10 shows the overall workflow. First, DURINN detects potentially racy two operation by analyzing a single-threaded memory trace. Second, if two racy operations are not consecutive, DURINN reorders the operations of the test case, places the two operations consecutively, and checks whether the same race can be triggered: *i.e.*, the new memory trace with the re-ordered operations still include the same race. Last, if two re-ordered operations are still racy, DURINN generates adversarial thread interleaving for these two operations. In the rest, we discuss each step in detail.

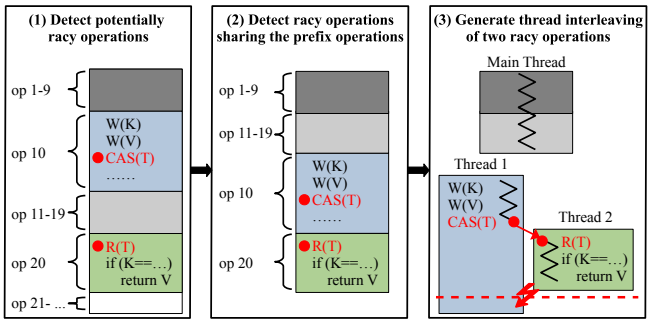


Figure 10: The workflow of adversarial NVM state and thread interleaving construction for Visible-But-Not-Durable bugs.

(1) Finding racy operations. The first step is to find potentially race operations. The inputs for the analysis are a single-threaded execution trace (§5.1) and the inferred likely-linearization points (§5.2). DURINN finds a pair of potentially racy operations that write-write or write-read synchronization variables (updated at likely-linearization points). These two potentially racy operations are not necessary to be consecutive in a single-threaded execution trace. In Figure 10 (1), operation 10 and 20 are such potentially racy operations.

(2) Finding prefix operations. A pair of potentially racy operations from the first step may not be racy when run in parallel. One main reason is that these two operations ran with different preceding operations (*i.e.*, prefix operations). In Figure 10 (1), the prefix of operation 10 is operation 1-9 but the prefix operations of operation 20 is operation 1-19. Hence, the precondition of an NVM data structure when running these two operations may be different, so these two operations may not be racy when run in parallel.

To filter out such spurious racy operations, DURINN reorders operations such that the two operations have the common prefix operations and places two potentially racy operations consecutively. In Figure 10 (2), operations 1-9 and 11-19 becomes the prefix of operation 10 and 20. We then run the instrumented program with the prefix and the two racy operations in a single thread and generate a new execution trace. If two candidates (operation 10 and 20) are still racy with the re-arranged operations, the prefix and racy operations will be fed in to the last step to construct thread interleaving of the two racy operations.

(3) Controlling thread interleaving and generating NVM state. For a given prefix operations, DURINN should precisely control thread interleaving of two concurrent racy operations. For example, in Figure 10 (3), thread 1 writes synchronization variable `T` first, which is LP, then thread 2 preempts and reads `T` then returns `V` to user. A crash should happen right after when the operation in thread 2 finishes and before thread 1 executes DP to trigger a Visible-But-Not-Durable bug.

In order to precisely control thread interleaving, DURINN uses a runtime technique using breakpoints. DURINN sets

breakpoints at the load and store of the synchronization variable (*e.g.*, `T` in Figure 10). After executing the prefix operations in a single-threaded manner, DURINN lets thread 1 run until reaching to the breakpoint of the `store` instruction to the synchronization variable. Then DURINN lets thread 2 run until it reaches the breakpoint of the `load` instruction of the synchronization variable. Then DURINN lets thread 2 resume and injects a crash right after finishing its operation.

Upon the crash, DURINN leaves all stores in thread 1 unpersisted as an adversarial NVM state. Note that thread 2 may not be able to finish its operation if other synchronization with thread 1 is involved (*e.g.*, deadlock). DURINN detects such a case with a timeout, and regards such thread interleaving infeasible. DURINN generates a `gdb` command script to automate the whole process. Using a hardware breakpoint makes DURINN's adversarial thread interleaving control efficent, though DURINN serializes a multi-threaded execution.

5.3.4 Cache and NVM Simulation

DURINN generates NVM crash images according to the adversarial NVM state and thread interleaving testing methods. To consider only feasible NVM states, DURINN simulates cache behaviors while obeying processor's memory consistency and persistence models. DURINN starts from the empty cache and NVM states and simulate the effects of `store`, `flush`, and `fence` instructions along an execution trace. Particularly, we implemented Intel's x86-64 architecture model following total store order (TSO) memory model consistency model [39, 56].

5.4 Durable Linearizability Validation

DURINN runs the NVM data structure under test from an NVM crash image (generated in §5.3) and checks if it violates durable linearizability by executing a sequence of validating operations. At a high level, the validating operations checks whether all operations before crash take effects (DL's C2 condition in §2.2); and whether the crashed operation is either fully executed or not at all executed (C3 condition). DURINN runs recovery code before running validation operations.

More specifically, for an NVM index data structure (*e.g.*, hash table, B-tree), the validating operations comprise:

1. A list of `get` operations to check all previously inserted but not deleted key-values exist,
2. A `get` operation to check the crash operations follows all or nothing semantics,
3. A list of `delete` operations for all inserted keys, and
4. A list of `get` operations to check all the deleted keys in the previous step are indeed deleted.

Note that for each crashed image, a list of completed operations and the crashing operation are known, so we know which key has been inserted or deleted. DURINN provide similar validating operations for other data structures: *e.g.*, array and queue.

6 Implementation

We implemented tracing and data flow analysis in LLVM [11]. We automatically generated `gdb` command files based on the locations of breakpoints. To control the progress of each thread in `gdb`, we set `scheduler-locking` on. Our LLVM-related code comprises around 1900 lines of C++ code. Other DURINN components are written in 2700 lines of Python code. Our current prototype supports an NVM program built on PMDK `libpmem` or `libpmemobj` libraries to create/load an NVM image from/to disk. To ensure the virtual address of the `mmap`-ed NVM heap are the same across different executions, we set `PMEM_MMAP_HINT` environment variable [38]. The DURINN prototype is available at https://github.com/cosmoss-jigu/durinn.

7 Discussion

7.1 False Negatives and False Positives in DURINN

DURINN may have false negatives (*i.e.*, missing bugs) for three reasons. First, DURINN is a trace-based dynamic tool that takes a test case as input. DURINN may miss DL bugs that did not appear in a trace.

Second, DURINN's likely-LP inference is based on heuristics and may miss true LPs in theory. Missing LP means no adversarial testing, so DURINN may miss DL bugs. However, the proposed heuristics are built on common NVM programming practices, namely Guarded-Protection and Publish-after-Initialization presented in §5.2. As a result, our empirical study (§8.4) shows that the inferred likely-LPs do not miss manually-identified (true/oracle) LPs and DURINN does not miss any DL bugs detected with the oracle LPs.

Last, DURINN performs adversarial testing and does not explore all possible NVM states and thread interleaving. In theory, for Incompletely-Durable and Unrecovered-Durable bugs, some more complex combinations of persisted and unpersisted stores may be required to trigger a DL bug. For Visible-But-Not-Durable bugs, more than two concurrent thread interleaving may be needed to expose a DL bug. However, our empirical study (§8.5) shows that DURINN detects all the bugs reported by the state-of-the-art Witcher [30] and indeed found more new bugs with significantly fewer tests.

On the other hand, for a given trace under test, DURINN does not have false positives as DURINN performs durable linearizability validation (§5.4). Any crash NVM image (constructed by adversarial testing) that violates durable linearizability is indeed a definite clue of a true DL bug (by definition). We note that multiple durable linearizability violations may stem from one root cause.

7.2 Persistent Cache

Intel architecture is expected to adopt eADR support (Extended Asynchronous DRAM Refresh) [37] that includes a cache into the persistent domain. For an eADR-enabled Intel architecture, there will be no gap between LP and DP because once the effect of a store reaches a cache, it is guaranteed to

be written back to the NVM.

We expect DURINN remain useful when eADR is available for the following three reasons. First, eADR is unlikely to be added to all product lines due to the high cost of battery size. In the current Intel platforms (Optane 200), eADR support is optional and requires an additional backup battery [36]. As eADR is not expected to be available in all machines, NVM programmers would need to write a code to support both eADR and non-eADR machines. Second, eADR is not the panacea if non-temporal stores (`ntstores`) are used. `Ntstores` place data in the store buffer and bypass caches, which is outside eADR persistent domain. For example, PMDK `pmemcpy()` prefers to use `movnstore` for better performance. An NVM program may lead to an inconsistent state when data in a store buffer is not flushed into NVM before a crash. Last, the Unrecovered-Durable bug pattern is still an issue even with eADR because it requires recovering or tolerating any partial updates made before linearization point. eADR has nothing to do with such a recovery logic. Developers still need to design and implement inconsistency-recoverable data structures.

7.3 Relationship to ACID

Three DL bug patterns can be disucssed using traditional database/filesystem's ACID terms. One can view *Incompletely-Durable* and *Unrecovered-Durable* bugs as ACID-atomicity/consistency/durability violations *Incompletely-Durable* bug pattern considers a crash after DP and tests ACID-atomicity/consistency/durability's fully-executed "all" semantic. *Unrecovered-Durable* bug pattern considers a crash before DP and tests ACID-atomicity's not-at-all-executed "nothing" semantic.

On the other hand, *Visible-But-Not-Durable* bug is related to ACID-isolation violation. *Visible-But-Not-Durable* bug pattern considers a crash before DP. However, a concurrent operation observed unpersisted data, and it completed, violating ACID-isolation and forcing the crashed operation to ensure the "all" semantic. A naive durability checker cannot detect *Visible-But-Not-Durable* bugs because they require a completed, ACID-isolation-violating, concurrent operation.

8 Evaluation

For evaluation, we first present our methodology (§8.1), then present the following experimental results.

- We report and analyze the DL bugs detected by DURINN (§8.2) along with detailed statistics, including the number of tests and testing time (§8.3).
- We evaluate the effectiveness and (empirical) soundness of DURINN's likely-linearization inference technique (§8.4).
- We compare DURINN with other NVM crash-consistency testing tools in terms of bug detection effectiveness and test space reduction (§8.5).

Application	Version	Type	Concurrency	Persistence
P-LF-BST [28]	5fa1dee	binary search tree	lock-free	LL
P-LF-Hash [28]	5fa1dee	hash table	lock-free	LL
P-LF-List [28]	5fa1dee	linked list	lock-free	LL
P-LF-Skiplist [28]	5fa1dee	skiplist	lock-free	LL
P-LF-Queue [29]	08fecfb	queue	lock-free	LL
CCEH [50]	d53b336	hash table	lock-based	LL
Fast Fair [34]	c86f5fb	B+ tree	lock-based	LL
P-ART [44]	5b4cf3e	radix tree	lock-based	LL
P-CLHT [44]	5b4cf3e	hash table	lock-based	LL
P-Hot [44]	5b4cf3e	trie	lock-based	LL
P-Masstree [44]	5b4cf3e	B tree + trie	lock-based	LL
pmdk-array [5]	v1.8	array	lock-based	LL
pmdk-queue [6]	v1.8	queue	lock-based	TX

LL: low-level persistence primitives **TX**: transactional persistence

Table 1: Tested concurrent NVM data structures

8.1 Evaluation Methodology

Tested NVM data structures. We evaluate DURINN with 13 concurrent NVM data structures, as listed in Table 1 with tested version, data structure type, its concurrency control mechanism, and persistence programming model. There are two concurrency control mechanisms: lock-free and lock-based. For persistence (durability) control, most data structures use low-level (LL) persistence primitives such as `flush` and `fence` instructions, while `pmdk-queue` uses PMDK's transactional (TX) persistence programming model.

All the tested data structures have been highly optimized for NVM, and most of them have shown to be more scalable than (simple) NVM hash tables and B-trees used in NVM-backed key-value stores such as `memcached`, `redis`, `pmemkv`, etc. All tested data structures use `libpmemobj`, the PMDK library for persistent memory allocation or transaction. As some data structures originally used a volatile memory allocator and emulated NVM using DRAM, we modified them to use the PMDK's persistent NVM memory allocator. Our changes do not add or delete any new/existing persistence primitives, or memory operations. Thus, the changes do not affect the bug detection evaluation.

Test Cases. We use AFL++ fuzzer [1] to generate a test case for our evaluation. We first feed a randomly generated seed into ALF++ fuzzer. Our random seed generator assigns a higher probability to create a new unused key for `insert`; and to reuse existing keys for other dependent operations such as `delete`, `update`, `query`. Then we run the fuzzer and picked the generated test case with the highest code coverage, which consists of 1,000 operations. We found that 1,000 operations are large enough to achieve a reasonable and stable code coverage (50%-80%) for our tested NVM data structures. Missing code coverage is due to unused features (*e.g.*, garbage collection) and debugging codes. The generated test cases are used for both likely-LP inference and adversarial testing.

Experimental setup. We ran all experiments on a 64-bit Fedora 29 machine with two 16-core Intel Xeon Gold 5218 processors (2.30GHz), 192 GB DRAM, and 512 GB NVM.

Name (Total #Bugs)	Bug ID	New	Confirm	Code	Type	Description	Impact	Fix strategy
P-LF-BST (1)	1	✓	✓	BSTAravindTraverse.h:331	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
P-LF-Hash (1)	2	✓	✓	ListTraverse.h:212	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
P-LF-List (1)	3	✓	✓	ListTraverse.h:212	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
P-LF-Skiplist(1)	4	✓	✓	SkiplistTraverse.h:218	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
P-LF-Queue(1)	5	✓	✓	DurableQueue.h:L74	DL1	Missing persistence primitives	Points to garbage	add persistence primitives
CCEH (2)	6	✓		CCEH_MSB.cpp:280	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	7		✓	CCEH_MSB.cpp:103	DL2	Atomicity in rehashing	Unable to recover	inconsistency-recoverable design
FAST-FAIR (5)	8	✓	✓	btree.h:955,979	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	9	✓	✓	btree.h:955,1007	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	10		✓	btree.h:224	DL1	Missing persistence primitives	Lost key-value	add persistence primitives
	11		✓	btree.h:213	DL2	Partial inconsistency is never recovered	unable to recover	inconsistency-recoverable design
	12		✓	btree.h:576	DL2	Atomicity in node splitting	unable to recover	logging/transaction
P-ART (4)	13	✓		Tree.cpp:35,258	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	14	✓		Tree.cpp:35,384	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	15		✓	N16.cpp:15	DL2	Atomicity between metadata and key-value	Unable to recover	inconsistency-tolerable design [8]
	16		✓	N4.cpp:17	DL2	Atomicity between metadata and key-value	Unable to recover	inconsistency-tolerable design [8]
P-CLHT (3)	17	✓		clht_lb_res.c:315,370	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	18	✓		clht_lb_res.c:315,468	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	19		✓	clht_lb_res.c:166	DL1	Missing persistence primitives	Lost key-value	add persistence primitives [9]
P-HOT (4)	20	✓		HOTRowex.hpp:61,84	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	21		✓	TwoEntriesNode.hpp:30	DL1	Missing persistence primitives	Points to garbage	add persistence primitives [9]
	22		✓	HOTRowexNode.hpp:315	DL1	Missing persistence primitives	Points to garbage	add persistence primitives [9]
	23		✓	HOTRowex.hpp:270	DL1	Missing persistence primitives	Points to garbage	add persistence primitives [9]
P-Masstree (3)	24	✓		masstree.h:1837,744	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	25	✓		masstree.h:1837,941	DL3	Incorrect concurrency control	Lost key-value	fix concurrency control/help persist
	26		✓	masstree.h:1378	DL2	Atomicity in node splitting	Unable to recover	logging/transaction
pmdk-array (1)	27		✓	array.c:486	DL2	Atomicity between metadata and data	Unable to recover	logging/transaxtion

DL1: Incompletely-Durable **DL2**: Unrecovered-Durable **DL3**: Visible-But-Not-Durable

Table 2: List of Durable Linearizability bugs detected by DURINN. In total, 27 (15 new) durable linearizability bugs were detected from 12 NVM data structures. There were 10 Incompletely-Durable bugs, 7 Unrecovered-Durable bugs, and 10 Visible-But-Not-Durable bugs.

App	# stores	# LPs	# DL1 tests	# DL2 tests	# DL3 tests	Execution time
P-LF-BST	10086	656	656	656	46	1m26s
P-LF-Hash	4604	547	547	547	5	1m44s
P-LF-List	4604	547	547	547	1623	7m15s
P-LF-Skiplist	26692	1040	1040	1040	491	4m3s
P-LF-Queue	9710	2000	2000	2000	7155	39m45s
CCEH	3631	1280	1280	1280	37	1m36s
Fast Fair	12989	10599	10599	10599	1585	8m37s
P-ART	12553	1112	1112	1112	287	2m34s
P-CLHT	2885	711	711	711	55	2m6s
P-HOT	32600	640	640	640	420	3m35s
P-Masstree	1403	1058	1058	1058	984	4m58s
pmdk-array	20505	3097	3097	3097	0	4m14s
pmdk-queue	57000	3000	3000	3000	0	2m51s
Total	199262	26287	26287	26287	12688	1h23m18s

DL1: Incompletely-Durable **DL2**: Unrecovered-Durable
DL3: Visible-But-Not-Durable

Table 3: The detailed statistics of DURINN bug finding.

8.2 Detected Durable Linearizability Bugs

In summary, DURINN detected 27 (15 new) durable linearizability bugs from 12 NVM data structures. There were 10 Incompletely-Durable bugs, 7 Unrecovered-Durable bugs and 10 Visible-But-Not-Durable bugs. 7 out of 15 new bugs have been confirmed by the developers so far. Table 2 shows the source code locations, impacts and fix strategies of the detected bugs.

(DL1) Incompletely-Durable bugs. DURINN detected 10 Incompletely-Durable bugs. Figure 6(a) discussed in §3.2 is a representative example (Bug ID 19) found in P-CLHT, leading to a lost key-value. As another instance, in P-LF-List (Bug ID 3), a new node is not fully persisted before it is added to the list using a `CAS` operation (which is LP). If a crash happens before DP (and after LP in this particular case), the list may contain a garbage node leading to an inconsistent structure. To fix Incompletely-Durable bugs, developers need to persist all the changes using additional cache line flush and fence instructions before DP.

(DL2) Unrecovered-Durable bugs. DURINN detected 7 Unrecovered-Durable bugs. Figure 6(b) illustrates a case detected in Fast-Fair (Bug ID 12). For another example, in CCEH (Bug ID 7), if a crash happens while rehashing the table and before adding a new segment into the table, the hash table will be in an illegal state: *i.e.*, all the metadata assumes there is a new segment added but it is not. To fix Unrecovered-Durable bugs, an NVM data structure should be able to recover from or tolerate partial updates before LP of an operation. Designing an inconsistency-recoverable design is one solution. Using logging or transaction is another.

(DL3) Visible-But-Not-Durable bugs. DURINN detected 10 Visible-But-Not-Durable bugs. Figure 6(c) shows a Visible-But-Not-Durable bug in Fast-Fair (Bug ID 8). For another example, Bug ID 6 from CCEH is due to incorrect usage of locks. While both `insert` and `get` operations use a lock to protect a critical section, the write to the synchronization variable (LP) is inside the critical section but the persistence of the synchronization variable (DP) is ensured outside the critical section in `insert`. Since the DP is not protected by a lock, the `get` operation is able to observe the visible but not durable writes from a concurrent `insert` operation. We observed two ways to fix Visible-But-Not-Durable bugs. Some

choose to fix the concurrency control mechanism to guarantee that every data read by concurrent threads is persisted. Others made one operation that reads unpersisted data help persist the data on behalf of another concurrent operation.

8.3 Statistics of DL Bug Detection

Table 3 shows the detailed statistics of DURINN when tested with 1000 operations. The second column reports the number of stores and the third column lists the number of inferred likely linearization points. On average, using static analysis described in §5.2, DURINN infers about 2,000 likely-LPs, which is 13% of 15.3K NVM stores traced while running 1000 tested operations. More detailed analysis on likely-LP inference will follow in §8.4.

The next three columns show the number of DL tests performed by DURINN to detect three DL bug patterns. The number of Incompletely-Durable and Unrecovered-Durable tests are the same as the number of inferred LPs because for each LP, DURINN performs one adversarial test for Incompletely-Durable bugs and for Unrecovered-Durable bugs. On the other hand, the number of Visible-But-Not-Durable tests depends on the number of co-schedulable racy operations. The second last column shows that the number of Visible-But-Not-Durable tests varies by data structures up to a few thousand. Intuitively, lock-free data structures tend to have more co-schedulable racy operations than (coarse-grained) lock-based ones, requiring more tests. The last column reports the execution time, which mostly depends on the number of tests. Testing all three test cases typically takes a few minutes. `P-LF-Queue` took the most time (around 40 mins) due to the large number of concurrent Visible-But-Not-Durable testing.

Lastly, for each test case violating durable linearizability, we manually analyze each case and report the details in Table 2. DURINN provides sufficient information for root cause analysis, including execution trace, crash location, persisted and unpersisted writes, and a crash NVM image. We loaded the crash image in `gdb` and followed the DURINN-generated schedule to inspect the root causes of detected DL bugs.

8.4 Likely-Linearization Point Inference

DURINN infers likely-linearization points using *Guarded-Protection* and *Publish-after-Initialization* heuristics described in §5.2. The number of likely-LP determines the number of DL tests that DURINN performs, so in terms of scalability, the less the better. At the same time, ideally, likely-LPs should not miss true LPs because missing LPs may lead to missing true DL bugs (false negatives).

We performed a detailed case study with CCEH and Fast-Fair in which we manually analyzed the true LPs (oracle) for comparison. They both use lock-based concurrency control in which the store instructions serving as LPs are not explicit. They are non-trivial concurrent data structures including balancing operations such as rehashing (CCEH) and node split/merge (Fast-Fair) operations.

Figure 11 shows the effectiveness of the proposed likely-LP inference techniques, compared to the manually identified LPs. The first bar represents the number of total stores. The second and third bar represent the number of likely-LPs when only Guarded-Protection or Publish-after-Initialization heuristics is used, respectively. The fourth bar shows the number of likely-LPs of DURINN where both are considered. The last bar is the number of LPs from our manual source code analysis. The result shows that DURINN effectively reduces the number of likely-LPs using two heuristics. The number of likely-LPs inferred by DURINN is twice as the number of manually-identified LPs. Note that as listed in Table 3, CCEH and Fast Fair are the most difficult data structures in terms of the reduction ratio between the stores and the likely-LPs.

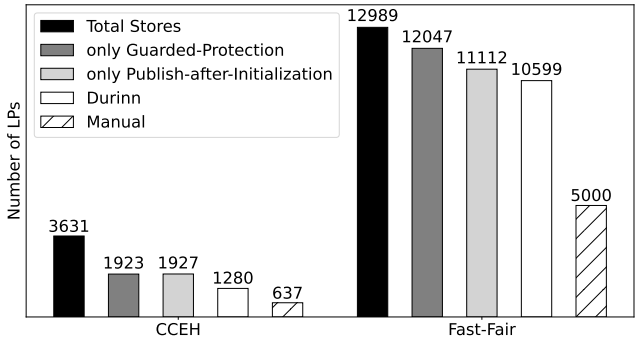


Figure 11: A case study of likely-linearization point inference.

Additionally, we compared the bug detection effectiveness and found that DURINN's inferred likely-LPs detect the same DL bugs as manually-identified (true) LPs. Though DURINN's likely-LP inference heuristics do not guarantee soundness in theory, this experiment empirically shows that likely-LP inference did not miss true LPs (at least) for the CCEH and Fast-Fair. We believe the same case for other data structures given that the heuristics are designed based on common NVM programming patterns.

8.5 Comparison with Other Tools

We present the detailed comparison with Witcher [30], the state-of-the-art NVM crash-consistency bug detector, and Yat [43], an exhaustive crash-consistency testing tool.

Bug Detection. We compared the bug detection effectiveness with Witcher. In their paper, Witcher claims that it can detect all the crash-consistency bugs that prior tools (*e.g.*, PMTest, XFDetector and Agamotto) found for a common set of NVM programs, along with some new bugs. For comparison, we run Witcher with the same test case with 1000 operations for six common data structures: CCEH, Fast-Fair, P-ART, P-CLHT, P-HOT and P-Masstree. Both Witcher and DURINN detected 11 bugs in common. Beyond them, DURINN reports 10 Visible-But-Not-Durable bugs that Witcher missed. Detecting Visible-But-Not-Durable bugs requires scheduling concurrent operations, which is not supported by Witcher.

Test Space Reduction. We compare the number of tests

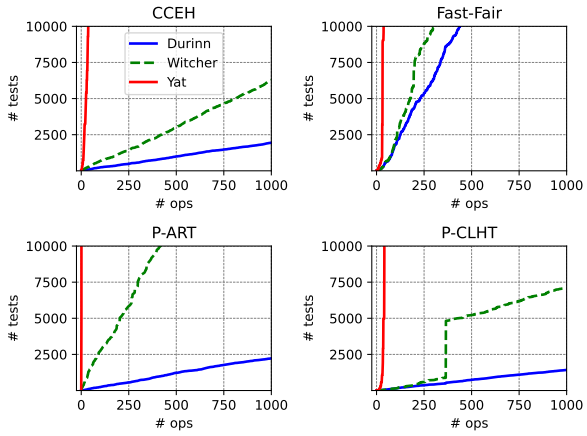


Figure 12: Test space comparison.

performed by DURINN, Yat, and Witcher over the same 1000 operations. Figure 12 shows how the number of tests grows (y-axis) as the number of tested operations increases (x-axis) for four data structures: CCEH, Fast Fair, P-ART, and P-CLHT. The other NVM data structures demonstrate a similar pattern. Yat is an exhaustive testing tool, so the test space explodes within the first ten operations. Witcher performs several times more tests than DURINN. The sudden spike in P-CLHT is due to a rehashing operation. On the other hand, DURINN performs adversarial testing for three DL bug patterns, reduces the number of tests, yet still detects more bugs than Witcher.

9 Related Work

File system/database Consistency Checking. File system consistency checking [17, 18, 32, 41, 48, 49, 54, 57, 60–62] deals with block-grained files, and logging/journaling is the norm for crash consistency. On the other hand, DURINN deals with concurrent data structures backed by byte-addressable NVM. DURINN analyzes load/store instructions along with cacheline flush and fence instructions, controlling durability in NVM. NVM data structures often come with custom (log-free) crash consistency and lock-free logic, making NVM test space huge. For file systems, CrashMonkey [49] bounds search space using heuristics learned from a bug study. EXPLODE [61] and FiSC [62] use in-situ model checking. In contrast, DURINN reduces test space with adversarial crash state and thread interleaving construction.

For correctness conditions, strict serializability and durable linearizability are equivalent from the ACID properties perspective. (PMDK "transaction" does not provide "isolation", though). Yet, we believe durable linearizability is the appropriate framework to use as many NVM data structures are derived from volatile concurrent data structures, where linearizability is the norm. Others [28, 42] also use durable linearizability. Equivalently, the asynchronous commit mechanism in database systems (Salt [59] and Hekaton [23]) can be mapped to "buffered durable linearizability" [40]. For performance, both do not eagerly make the changes durable as long as they can resume from one of the old consistent states.

Linearizability Checker. Line-up [15] is the first complete and automatic checker for deterministic linearizability. It detects thread-safety violations by comparing the concurrent execution to linearizable executions of a test. Similarly, Round-up [63] checks quasi linearizability. Quasi linearizability intentionally introduces non-determinism into the parallel computations and exploits such non-determinism to improve the performance. Pradel *et al.* [53] detects concurrency bugs in thread-safe classes. It generates tests in which multiple threads call methods on a shared instance of the tested class and check if the execution matches any linearizable execution.

Bug Detector for NVM Software. Most existing NVM bug detectors are not designed for durable linearizability bugs. PMDebugger [22] targets universal bugs such as missing persistence primitives. To detect application-specific bugs, such as persistence ordering/atomicity bugs, Yat [43], PMTest [46], XFDetector [45] and Agamotto [51] require user-defined custom oracles or consistency checkers. Jaaru [31] only identifies bugs that have visible manifestation, such as a segment fault or an assertion failure. Witcher [30] leverages all or nothing semantics for validation like DURINN, but it is limited to single-threaded NVM programs.

Active Testing. AtomFuzzer [52] is a randomized active atomicity violation detector, which modifies the thread scheduler behavior to create atomicity violations with high probability. RaceFuzzer [55] uses potential data race information obtained from an existing dynamic analysis technique to control a random scheduler of threads for actively detecting race conditions. Jumble [27] uses adversarial memory to classify race conditions as destructive or benign on systems with relaxed memory models. Relaxer [16] detects sequential consistency violations in a relaxed memory model by actively leading execution to predicted violations.

10 Conclusion

We present DURINN, the first durable linearizability checker for concurrent NVM data structures. We explore the gap between linearizability point and durability point, and define three novel durable linearizability bug patterns. We propose adversarial crash state and thread interleaving construction and likely-linearization point inference to detect durable linearizability bugs in an active and scalable manner. DURINN detected 27 (15 new) durable linearizability bugs.

Acknowledgments

We thank the anonymous reviewers and Murat Demirbas (our shepherd) for their insightful comments and feedback. This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant from the Korean government (MSIT) (No. 2014-3-00035) and by the National Science Foundation under the grants CNS-2029720 and CCF-2153747.

References

[1] American Fuzzy Lop plus plus (AFL++). URL: https://github.com/AFLplusplus/AFLplusplus.

[2] Argonne National Lab's Aurora Exascale System. URL: https://www.intel.com/content/www/us/en/customer-spotlight/stories/argonne-aurora-customer-story.html.

[3] Available first on Google Cloud: Intel Optane DC Persistent Memory. URL: https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud_intel-optane-dc-persistent-memory.

[4] Key/Value Datastore for Persistent Memory. URL: https://github.com/pmem/pmemkv.

[5] Persistent array in PMDK. URL: https://github.com/pmem/pmdk/tree/stable-1.8/src/examples/libpmemobj/array.

[6] Persistent queue in PMDK. URL: https://github.com/pmem/pmdk/tree/stable-1.8/src/examples/libpmemobj/queue.

[7] Pmem-Memcached. https://github.com/lenovo/memcached-pmem.

[8] RECIPE commit to fix reported bugs. URL: https://github.com/utsaslab/RECIPE/commit/4b0c27674ca7727195152b5604d71f47c0a0a7a2.

[9] RECIPE commit to fix reported bugs. URL: https://github.com/utsaslab/RECIPE/commit/950ae0ea5ed23ce28840615976e03338b943d57a.

[10] Redis v3.2. https://github.com/pmem/redis/tree/3.2-nvml.

[11] The LLVM Compiler Infrastructure. URL: https://llvm.org/.

[12] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. Bbb: Simplifying persistent programming using battery-backed buffers. In *Proceedings of the 27rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 111–124, Seoul, South Korea, February 2021.

[13] Anandtech. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here!, 2018. URL: https://www.anandtech.com/show/12828/intel-launches-optane-dimms_up-to-512gb-apache-pass-is-here.

[14] ARM Limited. ARM architecture reference manual armv8, for armv8-a architecture profile, 2020.

[15] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 330–340, New York, NY, USA, 2010. Association for Computing Machinery. URL: https://doi.org/10.1145/1806596.1806634, doi:10.1145/1806596.1806634.

[16] Jacob Burnim, Koushik Sen, and Christos Stergiou. Testing concurrent programs on relaxed memory models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 122–132, Toronto, Canada, July 2011.

[17] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 270–286, 2017.

[18] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37, 2015.

[19] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[20] CXL Consortium. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.

[21] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[22] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible and comprehensive bug detection for persistent memory programs extended abstract. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

[23] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD/PODS Conference*, pages 1243–1254, New York, USA, June 2013. ACM.

[24] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 478–493, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3341301.3359637, doi:10.1145/3341301.3359637.

[25] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[26] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, April 2014.

[27] Cormac Flanagan and Stephen N Freund. Adversarial memory for detecting destructive races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 244–254, Toronto, Canada, June 2010.

[28] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. NVTraverse: In NVRAM Data Structures, the Destination Is More Important Than the Journey. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 377–392, Virtual, June 2020.

[29] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, wien, Austria, March 2018.

[30] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, pages 100–115, Virtual, October 2021.

[31] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

[32] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 14:1–14:16, 2008.

[33] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. In *Proceedings of the ACM Transactions on Programming Languages and Systems*, pages 463–492, 1990.

[34] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 187–200, Oakland, California, USA, February 2018.

[35] Intel. pmreorder, 2019. URL: https://pmem.io/pmdk/manpages/linux/master/pmreorder/pmreorder.1.html.

[36] INTEL. Third Generation Intel® Xeon® Processor Scalable Family Technical Overview, 2020. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-xeon-processor-scalable-family-overview.html.

[37] Intel. eADR: New Opportunities for Persistent Memory Applications, 2021. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent_memory-applications.html.

[38] INTEL. PMDK man page: libpmem - persistent memory support library, 2021. URL: https://pmem.io/pmdk/manpages/linux/v1.0/libpmem.3.html.

[39] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2021. https://software.intel.com/en-us/articles/intel-sdm.

[40] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the 30st International Conference on Distributed Computing (DISC)*, pages 313–327, Paris, France, September 2016.

[41] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 147–161, 2019.

[42] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *SOSP '21: 28th ACM Symposium on Operating Systems Principles, October 25-28, 2021*. ACM, 2021.

[43] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, Philadelphia, PA, June 2014.

[44] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[45] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 1187–1202, Lausanne, Switzerland, April 2020.

[46] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 411–425, Providence, RI, April 2019.

[47] Micro. 3D XPoint Technology, 2019. URL: https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[48] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377, 2015.

[49] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 33–50, Carlsbad, CA, October 2018.

[50] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, Boston, MA, February 2019.

[51] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/neal.

[52] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 135–145, Atlanta, GA, November 2008.

[53] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. *SIGPLAN Not.*, 47(6):521–530, June 2012. URL: https://doi.org/10.1145/2345156.2254126, doi:10.1145/2345156.2254126.

[54] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–280, Dublin, Ireland, June 2009.

[55] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21, Tucson, AZ, June 2008.

[56] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[57] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2016.

[58] Steve Scargall. Programming Persistent Memory: A Comprehensive Guide for Developers, 2020. https://pmem.io/book/.

[59] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining acid and base in a distributed database. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[60] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[61] Junfeng Yang, Can Sar, and Dawson Engler. explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 10–10, Seattle, WA, November 2006.

[62] Junfeng Yang, Paul Twohey, and Dawson. Using model checking to find serious file system errors. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–288, San Francisco, CA, December 2004.

[63] Lu Zhang, Arijit Chattopadhyay, and Chao Wang. Round-up: Runtime checking quasi linearizability of concurrent data structures. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, page 4–14. IEEE Press, 2013. URL: https://doi.org/10.1109/ASE.2013.6693061, doi:10.1109/ASE.2013.6693061.