Detecting Persistence Bugs from Non-volatile Memory Programs by Inferring Likely-correctness Conditions

Xinwei Fu

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Ali R. Butt, Co-chair
Changwoo Min, Co-chair
Dongyoon Lee
Danfeng Yao
Xun Jian

February 10, 2022
Blacksburg, Virginia

Keywords: Non-Volatile Memory, Crash Consistency, Durable Linearizability,
Testing, Debugging
Copyright 2022, Xinwei Fu

Detecting Persistence Bugs from Non-volatile Memory Programs by Inferring Likely-correctness Conditions

Xinwei Fu

(ABSTRACT)

Non-volatile main memory (NVM) technologies are revolutionizing the entire computing stack thanks to their storage-and-memory-like characteristics. The ability to persist data in memory provides a new opportunity to build crash-consistent software without paying a storage stack I/O overhead. A crash-consistent NVM program can recover back to a consistent state from a persistent NVM in the event of a software crash or a sudden power loss. In the presence of a volatile cache, data held in a volatile cache is lost after a crash. So NVM programming requires users to manually control the durability and the persistence ordering of NVM writes. To avoid performance overhead, developers have devised customized persistence mechanisms to enforce proper persistence ordering and atomicity guarantees, rendering NVM programs error-prone. The problem statement of this dissertation is how one can effectively detect persistence bugs from NVM programs. However, detecting persistence bugs in NVM programs is challenging because of the huge test space and the manual consistency validation required. The thesis of this dissertation is that we can detect persistence bugs from NVM programs in a scalable and automatic manner by inferring likely-correctness conditions from programs. A likely-correctness condition is a possible correctness condition, which is a condition a program must maintain to make the program crash-consistent. This dissertation proposes to infer two forms of likely-correctness conditions from NVM programs to detect persistence bugs. The first proposed solution is to infer likely-ordering and likely-atomicity conditions by analyzing program dependencies among NVM accesses. The second proposed

solution is to infer likely-linearization points to understand a program's operation-level behavior. Using these two forms of likely-correctness conditions, we test only those NVM states and thread interleavings that violate the likely-correctness conditions. This significantly reduces the test space required to examine. We then leverage the durable linearizability model to validate consistency automatically without manual consistency validation. In this way, we can detect persistence bugs from NVM programs in a scalable and automatic manner. In total, we detect 47 (36 new) persistence correctness bugs and 158 (113 new) persistence performance bugs from 20 single-threaded NVM programs. Additionally, we detect 27 (15 new) persistence correctness bugs from 12 multi-threaded NVM data structures.

Detecting Persistence Bugs from Non-volatile Memory Programs by Inferring Likely-correctness Conditions

Xinwei Fu

(GENERAL AUDIENCE ABSTRACT)

Non-volatile main memory (NVM) technologies provide a new opportunity to build crash-consistent software without incurring a storage stack I/O overhead. A crash-consistent NVM program can recover back to a consistent state from a persistent NVM in the event of a software crash or a sudden power loss. NVM has been and will further be used in various computing services integral to our daily life, ranging from data centers to high-performance computing, machine learning, and banking. Building correct and efficient crash-consistent NVM software is therefore crucial. However, developing a correct and efficient crash-consistent NVM program is challenging as developers are now responsible for manually controlling cacheline evictions in NVM programming. Controlling cacheline evictions makes NVM programming error-prone, and detecting persistence bugs that lead to inconsistent NVM states in NVM programs is an arduous task. The thesis of this dissertation is that we can detect persistence bugs from NVM programs in a scalable and automatic manner by inferring likely-correctness conditions from programs. This dissertation proposes to infer two forms of likely-correctness conditions from NVM programs to detect persistence bugs, *i.e.*, likely-ordering/atomicity conditions and likely-linearization points. In total, we detect 47 (36 new) persistence correctness bugs and 158 (113 new) persistence performance bugs from 20 single-threaded NVM programs. Additionally, we detect 27 (15 new) persistence correctness bugs from 12 multi-threaded NVM data structures.

Dedication

To my family:

My Mother, Ying Deng,

My Father, Fangping Fu,

My Wife, Yanchen Wang,

My daughters, Mia and Olivia.

Acknowledgments

Six years ago, I settled down at Blacksburg and started chasing my dream. Time brought a significant change to the world, but it never changed my passion. I am grateful that I have lived and worked in this peaceful and lovely town during my Ph.D. Blacksburg is my daughters' birthplace and is also where my dream starts. I will never forget my Ph.D. journey in Blacksburg, and I have lots of people to thank here.

First of all, I would like to express my most profound appreciation to my advisors, Drs. Dongyoon Lee and Changwoo Min. With their guidance and support, I found the beauty of research and enjoyed exploring the boundary of knowledge. Their valuable advice makes me a better researcher and also a better person. I would not be able to make such an achievement without their selfless advice, and I feel so fortunate to have them as my advisors. I sincerely hope that we will continue our collaborations in the future.

I would also like to thank my committee, Drs. Ali Butt, Danfeng Yao, and Jian Xun, for their constructive comments and feedback that improve this dissertation's overall quality. I am also grateful to my collaborators, Drs. Changhee Jung, Sudarsun Kannan, James C. Davis, Sanidhya Kashyap, Jaeho Kim, and Wook-Hee Kim, for their collaborative efforts in our joint research projects.

I am grateful to all the friends I have met and worked with over the past six years. They did inspire and improve me in different aspects. My Ph.D. journey has become colorful and cheerful because of them. I believe our friendship will be one of the most valuable fortunes for the rest of my life.

Sincerely, I would like to thank my parents for their selfless love and support. I grew up while my parents became old. Being abroad for six years, I have never fulfilled the responsibility as

a son. I owe my parents an immense debt, and I firmly believe I will realize their expectations of me in the future. I will treasure every moment with them ever since.

Lastly and significantly, I would like to thank my wife, Yanchen Wang, for everything! My wife is the bravest and most tenacious lady I have ever met in my life. She gave birth to our two daughters during my Ph.D. and devoted everything she had to our little family. Although life was so hard during COVID-19, we came through together. Difficulty or failure never beat us but only made the foundation of our love stronger. Words cannot express my love for her, and I will use the rest of my life to prove it.

Contents

List of Figures

List of Tables

List of Abbreviations

CAS Compare-and-swap

CXL Compute Express Link

DL Durable Linearizability

DP Durability Point

eADR Extended Asynchronous DRAM Refresh

EDA Event-driven Architecture

FASE Failure-atomic Critical Section

GPF Global Persistent Flush

ISA Instruction Set Architecture

LP Linearization Point

NVM Non-volatile Main Memory

SV Synchronization Variable

Chapter 1

Introduction

NVM technologies are being widely adopted in various existing and future computer systems. Notably, Intel's Optane DC Persistent Memory [15, 93] has already been deployed in Google Cloud [4] and Aurora supercomputers [2]. ARM also has announced its support for NVM [14, 16]. The upcoming Compute Express Link (CXL) [30] standard introduces a cache-coherent CXL-attached NVM card with an on-device cache.

NVM provides storage-and-memory-like characteristics [118]. Like storage, NVM is persistent and has a high capacity. Every data recorded in NVM is persistent thus can be accessed even after a system crash or power failure. The largest capacity of Intel's Optane Persistent Memory is 512GB per DIMM, while the capacity of DRAM typically ranges from 16GB to 64GB per DIMM. Like memory, NVM is byte-addressable and has low access latency. Users can directly access NVM using regular load and store instructions. The access latency of NVM is comparable to DRAM and is much faster than traditional block-based persistent storage disks.

The ability to persist data in memory provides a new opportunity to build crash-consistent software without paying storage stack overheads. An NVM program is *crash-consistent* if it can recover back to a consistent state from NVM in any crash event and seamlessly resume its execution. Durable linearizability [64] is the widely-used correctness standard that determines whether an NVM program is in a consistent state. Durable linearizability defines correct behaviors of NVM programs when crashes happen. Durable linearizability

extends the Linearizability model [50], the widely-used correctness standard for concurrent programs, with the notion of a crash. An NVM program is *durably linearizable* if all of its executions are linearizable once crash events are removed. We will have detailed discussions about durable linearizability later in §2.3.

However, it is hard to build a correct and efficient crash-consistent NVM program. In the presence of a volatile cache, NVM data held in a volatile cache is lost after a crash. A cache can also evict cachelines in an arbitrary order. Thus, the updates to different NVM locations may not be persisted in the same order as the program store order. Furthermore, existing instruction set architectures (ISAs) also do not support atomically updating multiple NVM locations.

NVM programming requires developers to manually control the durability and the persistence ordering of NVM writes. Two persistence primitives are provided to control cacheline evictions to support NVM programming. Data becomes persistent when evicted from a cache and reaches an NVM. The first primitive is `flush` (*e.g.*, `clwb` in Intel x86 architecture), which asynchronously writes back a cacheline from a cache to a memory. The other primitive is `fence` (*e.g.*, `sfence` in Intel x86 architecture), which guarantees the previous asynchronous flushes are completed. Using `flush` and `fence`, NVM program developers are able to control cacheline evictions to build crash-consistent NVM software without paying a storage stack I/O overhead. Due to the overhead of persistence primitives, `flush` and `fence` should be used as little as possible to achieve better performance. To use persistence primitives as little as possible, an NVM developer needs to create a crash-consistent design with a minimal number of persistence ordering and atomicity guarantees. Creating this type of design is error-prone. Thus, using a minimal number of `flush` and `fence` primitives to achieve crash-consistency is hard.

NVM program developers apply two different techniques to achieve crash-consistency. One

solution is to use general logging techniques, such as UNDO/REDO logging [24, 52, 56, 82]. The other solution is to use log-free persistence mechanisms, which use customized persistence ordering and atomicity to achieve crash-consistency. Logging techniques have an inherently high overhead since logging techniques lead to additional or redundant `flush` and `fence` instructions. For example, in UNDO logging, a log entry of the backup data must be persistent before updating the corresponding data, which costs extra memory overhead for storing this backup data and additionally incurs performance overhead to guarantee the persistence ordering and atomicity constraints. Log-free persistence mechanisms are fast but error-prone. Log-free persistence mechanisms take advantage of program-specific logic to use `flush` and `fence` primitives as little as possible to reduce persistence overhead. However, any misuse of persistence primitives may lead to incorrect persistence ordering or atomicity, thus potentially making an NVM program no longer crash-consistent.

Persistence bugs are classified into two categories: (1) persistence correctness bugs (breaking crash-consistency guarantees) and (2) persistence performance bugs (degrading program performance). A persistence correctness bug leads to an inconsistent NVM state on a crash and a failure to recover. This bug could cause unintentional permanent data corruption, irrecoverable data loss, etc. A persistence performance bug leads to unnecessary program performance degradation.

1.1 Problem Statement

Several solutions have been proposed to detect persistence bugs in NVM programs. However, those solutions fail to detect persistence bugs in a scalable and automatic manner due to the two critical issues: (1) the scalability issue when testing possible NVM states and thread interleavings and (2) the automation issue when validating correctness.

A line of testing tools [58, 78] attempts to exhaustively test all possible NVM states. The exhaustive search solution can detect many bugs but often suffers from test space explosion. For example, persistent indexes typically include rebalancing operations for a time complexity guarantee (e.g., rehashing in a hash table, split-merge in a B-tree). Triggering such operations and detecting persistence bugs therein requires a test case with many operations (a long execution), thus making exhaustive testing infeasible in practice.

Moreover, when using test oracles, existing NVM testing tools require users to provide a manually-designed, program-specific consistency checker [35, 58, 78, 84, 85, 98] to validate a program under test. NVM programs often employ different forms of tolerable inconsistency and/or recovery designs. Devising program-specific test oracles requires significant manual effort and is error-prone.

The problem statement of this dissertation is:

How can one scalably and automatically detect persistence bugs from NVM programs?

1.2 Thesis

The thesis of this dissertation is that:

We can detect persistence bugs from NVM programs in a scalable and automatic manner by inferring likely-correctness conditions from programs. Instead of testing all possible NVM states, we only test those NVM states violating likely-correctness conditions.

A likely-correctness condition is a possible correctness condition, which is a condition a program must maintain to make the program crash-consistent. This dissertation proposes to infer two forms of likely-correctness conditions to detect persistence bugs from NVM programs: (1) likely-ordering/atomicity conditions and (2) likely-linearization points.

A likely-ordering/atomicity condition describes a possible persistence ordering or atomicity guarantee among NVM accesses to make the program crash-consistent. For example, the write to NVM address `A` should be persisted before the write to NVM address `B`, or the writes to NVM addresses `A` and `B` should be persisted atomically. Likely-ordering/atomicity conditions are inferred from NVM programs by analyzing program dependency between NVM accesses.

A likely-linearization point is a possible linearization point, where an operation takes effect and its effects become visible to other operations. We can analyze the behaviors of an NVM program at the program operation level by using likely-linearization points. For example, when a crash happens after a certain point within an operation, the crashing operation should ensure that its effects remain visible after the crash. Likely-linearization points are inferred from source code based on the common concurrent NVM programming practices.

Using inferred likely-correctness conditions, we then test only those NVM states and thread interleavings that violate our likely-correctness conditions, significantly reducing the test space needed to be examined. However, a likely-correctness condition violation does not necessarily mean a persistence bug. A validation process is needed to check whether an NVM state is consistent or not. Prior works require users to provide manually-designed, program-specific test oracles for validation. To avoid creating manual test oracles, we leverage the durable linearizability model to validate consistency automatically. For example, the effects of completed operations before a crash should remain visible; the operations that have not been completed upon a crash could be considered either completed or not. In this way, we can detect persistence bugs from NVM programs in a scalable and automatic manner.

1.3 Contributions

This dissertation proposes to infer likely-correctness conditions from NVM programs to detect persistence bugs in a scalable and automatic manner. We designed and implemented two persistence bug detectors based on two forms of likely-correctness conditions: likely-ordering/atomicity conditions and likely-linearization points.

1.3.1 Likely-ordering/atomicity Conditions

We propose WITCHER, which infers likely-ordering/atomicity conditions to detect persistence correctness bugs in NVM programs in a scalable and automatic manner. WITCHER automatically infers *likely-ordering/atomicity conditions* by analyzing data and control dependencies among NVM accesses. WITCHER then validates if any violation is a true persistence correctness bug by checking *output equivalence* between executions with and without a crash. The evaluation with 20 NVM programs based on Intel's PMDK library shows that WITCHER discovers 47 (36 new) persistence correctness bugs without a test space explosion problem or needing any supplementary manual consistency checkers.

1.3.2 Likely-linearization Points

We propose DURINN, which infers likely-linearization points to detect persistence bugs in NVM programs in a scalable and automatic manner. DURINN is based on the novel observation of the gap between a linearization point – when the changes to a concurrent data structure become publicly visible – and a durability point – when the changes become persistent. From our detailed gap analysis, we derive three durable linearizability bug patterns that render a linearizable data structure not durably linearizable. To tame the huge test

space of NVM states and thread interleavings, Durinn statically identifies likely-linearization points, and then actively constructs adversarial NVM states and thread interleavings to increase the likelihood of revealing persistence correctness bugs. Durinn effectively detected 27 (15 new) persistence correctness bugs from 12 concurrent NVM data structures without a test space explosion problem or needing any supplementary manual consistency checkers.

1.4 Broader Impact

This dissertation aims to make NVM programming easy by providing debugging support for NVM programs. The long-term vision of this dissertation is to help our community to build bug-free NVM programs.

NVM technologies have become publicly available from April 2019. NVM is expected to be used in services integral to our daily life, ranging from data centers to high-performance computing, machine learning, and banking. Building correct and efficient crash-consistent NVM software is therefore crucial. It is, however, challenging for developers, many of whom may not be as experienced as NVM system developers. We believe our proposed research can effectively improve our understanding of NVM bugs, significantly reducing the burden in detecting potential bugs, and help developers build correct and efficient crash-consistent NVM programs more easily. We also believe our research would generate fruitful discussions and insights in academia and industry.

1.5 Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces the background of this dissertation. Chapter 3 introduces the motivation of this dissertation. Chapter 4

presents the overview of our proposed solutions. Chapter 5 presents the detailed design of WITCHER, which infers likely-ordering/atomicity conditions to detect persistence correctness bugs. Chapter 6 presents the detailed design of DURINN, which infers likely-linearization points to detect persistence correctness bugs. Chapter 7 presents the design of the trace-based persistence performance bug detection. Chapter 8 presents the implementation of our proposed solutions. Chapter 9 presents the evaluation of our proposed solutions. Chapter 10 discusses soundness, completeness, and extensions of our proposed solutions. Chapter 11 introduces related works of this dissertation in a broad scope. Chapter 12 concludes the dissertation.

Chapter 2

Background

This chapter presents the background of this dissertation. We first introduce non-volatile memory technology (§2.1), then discuss logging-based and log-free persistence mechanisms (§2.2) used for guaranteeing crash-consistency, and lastly present the Durable Linearizability model (§2.3) as the NVM correctness condition.

2.1 Non-volatile Memory

NVM technologies, such as the recently commercialized Intel Optane DC Persistent Memory [15, 93], provide persistence and high storage capacity along with traditional DRAM characteristics such as byte addressability and low access latency. NVM is persistent, so every data recorded in NVM can be accessed back after a system crash or power failure. NVM has a high capacity. The largest capacity of Intel Optane is 512GB per DIMM, while the capacity of DRAM typically ranges from 16GB to 64GB. NVM is byte-addressable, so users can directly access NVM using regular `load` and `store` instructions. NVM has low access latency comparable to DRAM and is much faster than block-based storage SSD. In addition, NVM offers lower energy consumption and larger capacity at significantly lower $/GB than DRAM [68, 89, 108, 119, 129].

In modern architecture, NVMs are attached to processors via a memory bus so that programs can access the NVMs using regular `load` and `store` instructions. The ability to directly

access NVMs provides a new opportunity to build *crash-consistent* software without paying storage stack overhead. Programs can recover back to a consistent state from a potentially-inconsistent persistent NVM state in the event of an application or system crash or a sudden power loss.

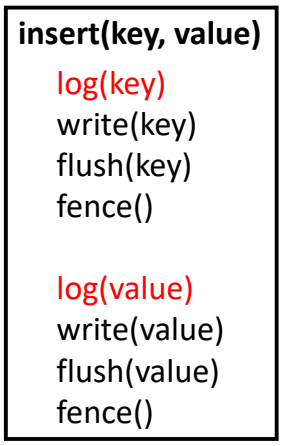
This dissertation assumes a volatile cache, which is the case for the current Intel x86 architecture with Optane NVM [62, 118] and ARM [14, 16]. The future Compute Express Link (CXL) [30] standard also introduces a cache-coherent interconnect and a CXL-attached NVM card with a volatile on-device cache.

Given a volatile cache, dirty cachelines are lost upon a crash. To control durability, Intel provides cache flush instructions such as `clflush`, `clflushopt`, and `clwb`. While the flush `clwb` instruction is asynchronous to reduce performance overheads, the store fence `sfence` instruction should be used together to ensure the completion of preceding `clwb` instructions [118]. Similarly, ARM supports the `dc cvap` cache flush and `dsb` fence instructions [14, 16]. CXL introduces Global Persistent Flush (GPF) to enforce the persistence ordering on emerging CXL-attached NVM card [30].

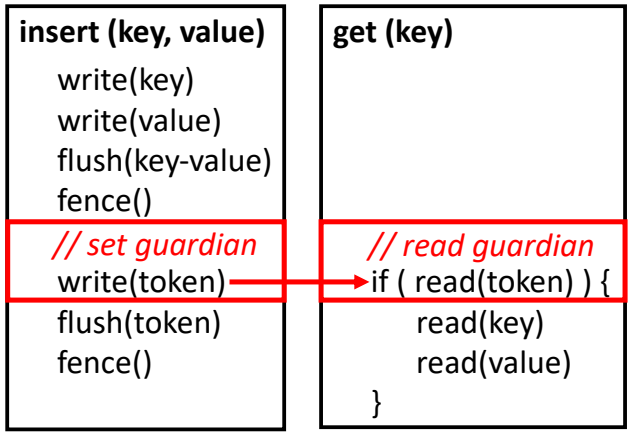
2.2 Logging-based and Log-free Persistence Mechanisms

There are mainly two types of mechanisms used for guaranteeing crash-consistency of NVM programs: logging-based and log-free persistence mechanisms.

Logging-based persistence mechanisms are general but have a high overhead. Logging is a general solution without considering program-specific logic and can be applied to any NVM program. It provides persistence atomicity for a logged region, *i.e.*, a transaction. If a crash happens within a transaction, this transaction should be either fully executed or not



(a) Logging-based persistence mechanism example: UNDO logging.



(b) Log-free persistence mechanism example: guarded protection.

Figure 2.1: Logging-based and log-free persistence mechanism examples.

at all executed after a recovery process. UNDO logging is the most widely used logging-based persistence mechanism for guaranteeing crash-consistency of NVM programs. With UNDO logging, when a crash happens within a transaction, every NVM update within that transaction before the crash will be undone after a recovery process. To undo NVM updates in a recovery process, UNDO logging requires that a log entry of the backup data must be persistent before updating the corresponding data. As shown in Figure 2.1(a), the key and value must be logged before writing the key and value. The additional persistence primitives and specific persistence ordering in UNDO logging lead to high overhead.

The other solution is to use log-free persistence mechanisms with low overhead but error-prone. To use persistence primitive as little as possible and still guarantee crash-consistency, log-free persistence mechanisms use program-specific logic to establish customized persistence ordering for each NVM program. The *guarded protection* [45] pattern is a widely-used log-free persistence mechanism to guarantee crash-consistency in low overhead. Figure 2.1(b) shows an example of the guarded protection pattern. The guarded protection pattern intends to ensure the atomic persistence of both key and value, *i.e.*, the result of the insertion is

either fully executed or not at all executed. Instead of using logging, the guarded protection pattern introduces a guardian variable, the *token* in this example. In this example, the writer writes the token after persisting the key and value. When setting the guardian, it ensures that the key and value have been already persisted. On the reader side, it checks whether the token has been set. If the token has been set, it means that the key and value have been already persisted. Then the reader can safely read the key and value. In summary, the guarded protection pattern guarantees that when the token is valid, the key and value must be persisted to achieve the atomicity. This way, the program-specific persistence ordering is used to achieve crash-consistency and low overhead instead of general logging persistence mechanisms. However, log-free persistence mechanisms are error-prone. Customized persistence ordering requires to use of persistence primitives. Any misuse of persistence primitives may lead to incorrect persistence ordering or atomicity, thus potentially making an NVM program no longer crash-consistent.

2.3 NVM Correctness Condition: Durable Linearizability

A crash-consistent NVM program can recover back to a consistent state from NVM in any crash event. However, how can one determine whether an NVM state is consistent? The existing widely-used NVM correctness condition is the *Durable Linearizability* model. Durable Linearizability defines correct behaviors of NVM programs when crashes happen. It extends the *Linearizability* model, the existing widely-used correctness standard for concurrent programs, with the notion of a crash. In this section, we present the Linearizability model first and then introduce the Durable Linearizability model.

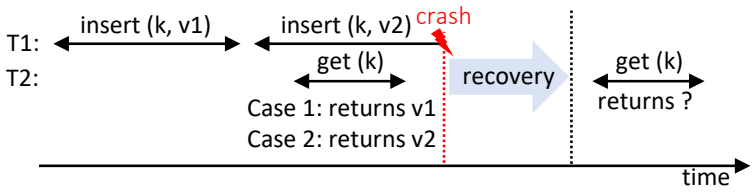


Figure 2.2: A durable linearizability example. The result of the second `get` (after a crash) depends on the result of the first `get` (before a crash), which also determines if the crashed `insert` takes effect.

Linearizability [50] 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) serves as the linearization point. In a lock-free data structure, the linearization point is typically a single-step atomic instruction (*e.g.*, `compare-and-swap` or simply `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`).

Durable linearizability [64] 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.

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 been 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 2.2 illustrates a durable linearizability example. Thread `T1`'s first `insert` completes before a crash, so it should be visible after a crash by (C2). For the same reason, if `T2`'s first `get` returns `v1` (or `v2`) before a crash, the second `get` after the crash should return the same `v1` (or `v2`). This implies that `T1`'s crashed second `insert` did not take effect and was not persisted for Case 1 (and took effect and was persisted for Case 2) according to the all or nothing semantics in (C3).

Chapter 3

Motivation

Developing correct and efficient crash-consistent NVM programs is error-prone. NVM data held in a volatile cache is lost after a crash. A cache can also evict cache lines in an arbitrary order. Thus, the updates to different NVM locations may not be persisted in the same order as the program (store) order. Existing architectures also do not support updating multiple NVM locations atomically. Current NVM programming models require developers to explicitly add persistence primitives to enforce proper persistence ordering and atomicity guarantees in order to ensure crash consistency. To achieve a low persistence overhead, developers are forced to use persistence primitives as little as possible. Any misuse of an NVM primitive may lead to persistence correctness bugs or persistence performance bugs. NVM programming thus becomes error-prone.

There are mainly two challenges to detect persistence bugs in a scalable and automatic fashion: (1) the scalability issue when testing possible NVM states and thread interleavings and (2) the automation issue when validating correctness. Existing persistence bug detectors are based on either *exhaustive testing* or *program annotation* and fail to detect persistence bugs in a scalable and automatic manner.

In this chapter, we present the motivation of this dissertation. We first present persistence correctness and performance bugs (§3.1), then introduce the challenges of detecting persistence bugs scalably and automatically (§3.2), and lastly discuss the limitations of existing persistence bug detectors (§3.3).

3.1 Persistence Bugs

Persistence bugs are classified into two categories: (1) persistence correctness bugs (breaking crash-consistency guarantees) (§3.1.1) and (2) persistence performance bugs (degrading program performance) (§3.1.2). A persistence correctness bug leads to an inconsistent NVM state on a crash and a failure to recover. A persistence performance bug leads to unnecessary program performance degradation.

3.1.1 Persistence Correctness Bugs

As a processor can evict cache lines in an arbitrary order and does not support atomic update of multiple NVM locations, crash consistency is the problem of guaranteeing persistence ordering and atomicity of NVM locations according to the program's semantics. Thus, violating these is a primary source of correctness bugs in NVM programs. Based on the root causes, persistence correctness bugs are classified into two categories: (1) persistence ordering bugs and (2) persistence atomicity bugs.

(1) Persistence ordering bugs. We found that many crash consistency and recovery mechanisms rely on a certain *persistence ordering* of NVM variables. For example, the write to NVM address `A` must be persisted before the write to NVM address `B`. However, a buggy NVM program may not maintain proper persistence ordering using a cacheline flush and a store fence instructions when updating multiple NVM locations.

(2) Persistence atomicity bugs. To ensure the integrity of NVM data, many NVM programs rely on atomic updates of multiple NVM variables. For example, the writes to NVM addresses `A` and `B` must be persisted persisted atomically. However, a buggy NVM program may not correctly enforce *persistence atomicity* among multiple NVM updates. If a program crashes in the middle of a sequence of updates, an inconsistent state may occur.

3.1.2 Persistence Performance Bugs

Previous studies [84, 98] found that persistence performance bugs are prevalent in real-world NVM programs. A persistence performance bug leads to unnecessary program performance degradation. Persistence performance bugs do not cause an inconsistent state, yet it requires significant developers' time and effort to spot and fix them. Similar to prior work, we classify persistence performance bugs as follows:

(1) Unpersisted performance bugs. Some NVM programs unnecessarily place volatile data that does not require persistence in NVM. Developers do not use flush/fence for volatile data. However, NVM accesses have higher latency than DRAM accesses. Developers should have placed them in DRAM.

(2) Extra flush and (3) extra fence performance bugs. An extra flush or fence instruction on an NVM variable causes unnecessary high overhead. Removing the extra ones does not break the correctness of an NVM program.

(4) Extra logging performance bugs. When an NVM program relies on a transaction library (*e.g.*, Intel's PMDK) for crash consistency, the NVM data should be (undo) logged before it is modified the first time. Logging the same NVM region redundantly in a transaction is a performance bug.

3.2 Challenges of Detecting Persistence Bugs

To scalably and automatically detect persistence bugs, we must address the following two challenges:

(1) The scalability problem against huge test space. The testing space grows expo-

nentially in two dimensions: *NVM crash states* and *thread interleavings*. The crash state test space is huge since a crash may happen any time during execution, and a volatile cache can evict cachelines in an arbitrary order. For example, Yat [78], an exhaustive crash consistency testing tool, attempts to test 10^{31} crash states for an NVM hash table with 2000 operations [45]. Moreover, the number of thread interleavings grows exponentially (n^k) with the number of threads (n) and the number of steps (k) in each thread.

(2) The need to create manual test oracles to validate correctness. Validating the correctness of each test requires test oracles. Manually creating test oracles is not automatic and is also error-prone, especially for NVM programs using log-free persistence mechanisms. Since log-free persistence mechanisms are based on program-specific logic, each NVM program's persistence ordering and atomicity guarantees are also different. To validate correctness, developers need to provide program-specific test oracles that require significant manual efforts and are error-prone.

3.3 Limitations of Existing Persistence Bug Detectors

Existing solutions are based on either *exhaustive testing* or *program annotation* to detect persistence bugs in NVM programs. However, two critical issues are (1) the scalability issue when testing possible NVM states and interleavings and (2) the need to create manual test oracles to validate correctness. These issues make existing solutions fail to detect persistence bugs in a scalable and automatic manner. Table 3.1 summarizes how WITCHER and DURINN are different from existing persistence bug detectors when detecting persistence correctness bugs.

Exhaustive testing. Exhaustive testing tools, such as Yat [78] and PMReorder [58], attempt to permute all possible NVM states on a crash. However, they often do not scale. For example, during testing Level Hashing with 2000 operations, Yat attempts to test 10^{31} total

	Test space exploration		Crash consistency validation (oracle)
	Input	**NVM state & Thread interleave**	
Yat [78] PMReorder [58]	user-provided test case	exhaustive	user-provided oracle
Jaaru [47]	user-provided test case	model checking with pruning	visible manifestation
PMTest [84] XFDetector [85]	user-provided test case	manual annotation	user-provided oracle
Agamotto [98]	symbolic execution	PM-aware search algorithm	user-provided oracle
PMDebugger [35]	user-provided test case	user-provided oracle	user-provided oracle
Witcher Durinn	user-provided test case	pruning based on likely-correctness conditions	durable linearizability validation

Table 3.1: Comparison with existing persistence bug detectors.

permutations. Moreover, they rely on a user-provided consistency checker for each crashed state to validate whether NVM data is consistent. However, the correctness of a manual checker is often a concern [65]. Recently, Jaaru [47], a model checking approach, proposed a (sound) state pruning solution based on the actual values read by post-failure executions, yet the test space may remain huge. Empirically, Jaaru has been applied to the test cases with up to (small) 40 operations. In addition, Jaaru can only identify bugs that lead to visible crashes (*e.g.*, segmentation faults) or assertion failures.

Annotation-based approach. The test space explosion problem motivated the *annotation-based approach*, such as PMTest [84] and XFDetector [85]. However, annotating a large NVM software soundly and precisely is very challenging. A missing/wrong annotation may lead to false negatives/positives. In addition, PMTest lacks support for detecting persistence atomicity bugs. XFDetector relies on users' manual investigation for validation. Agamotto [98] takes a different approach, using symbolic execution to explore input test space (program paths). It provides universal bug oracles for common bug patterns (*i.e.*, missing or redundant flush/fence bug patterns). However, for log-free (program-specific) persistence correctness bugs (*e.g.*, persistence ordering/atomicity bugs), Agamotto still requires users to provide test oracles. Similarly, PMDebugger [35] requires user-provided oracles (*i.e.*, ordering debugger configuration file) to detect log-free (program-specific) correctness bugs.

Chapter 4

Overview of Proposed Solutions

This dissertation proposes likely-correctness condition inference to detect persistence bugs in a scalable and automatic manner.

A likely-correctness condition is a possible correctness condition, which is a condition a program must maintain to make the program crash-consistent. Using inferred likely-correctness conditions, we test only those NVM states and thread interleavings that violate likely-correctness conditions, significantly reducing the test space. After that, we leverage the durable linearizability model to validate correctness automatically without creating manual test oracles. A likely-correctness condition does not need to be a true correctness condition. If two likely-correctness conditions violate each other, we test both of them and then validate those two tests based on the durable linearizability model.

Figure 4.1 illustrates our proposed solutions' architecture that takes a target program and a test case as input and reports detected persistence correctness and persistence performance bugs in the program as output. We first instrument the program and run the test case to collect a memory trace. For persistence correctness bugs, we infer likely-correctness conditions from the trace, construct a set of NVM states and thread interleavings violating the likely-correctness conditions, and perform durable linearizability validation to validate if a likely-correctness condition violation is a true persistence correctness bug. We analyze the same trace to detect persistence performance bugs by using a trace-based solution.

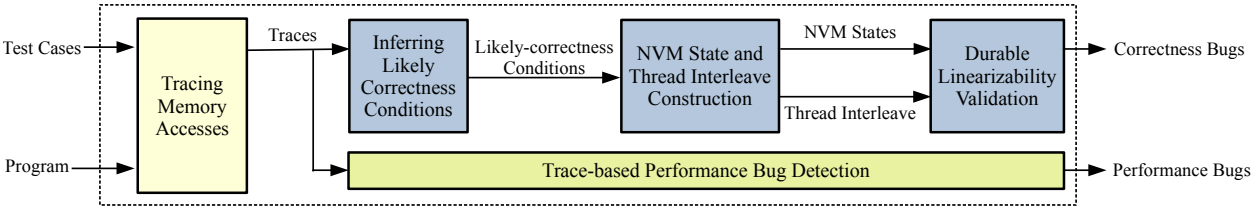


Figure 4.1: The architecture of proposed solutions. Our proposed solutions automatically detect persistence correctness bugs (in blue) and persistence performance bug (in green) based on a given test case and its trace without either manual annotation/oracle or exhaustive testing.

This dissertation proposes two forms of likely-correctness conditions for persistence bug detection: the likely-ordering/atomicity condition at the instruction level and the likely-linearization point at the operation level.

A likely-ordering/atomicity condition describes a possible persistence ordering or atomicity guarantee among NVM accesses to make the program crash-consistent. For example, the write to NVM address `A` should be persisted before the write to NVM address `B`, or the writes to NVM addresses `A` and `B` should be persisted persisted atomically. Likely-ordering/atomicity conditions are inferred from NVM programs by analyzing program dependency between NVM accesses.

A likely-linearization point is a possible linearization point, where an operation takes effect and its effects become visible to other operations. We can analyze the behaviors of an NVM program at the program operation level by using likely-linearization points. For example, when a crash happens after a certain point within an operation, the crashing operation should ensure that its effects remain visible after the crash. Likely-linearization points are inferred from source code based on the common concurrent NVM programming practices.

This chapter first introduces how to infer likely-ordering/atomicity conditions (§4.1) and likely-linearization points (§4.2) from NVM programs for persistence correctness bug detection, then presents the trace-based persistence performance bug detection (§4.3).

4.1 Likely-ordering/atomicity Condition Inference

We propose WITCHER, which detects persistence correctness bugs by inferring likely-ordering/atomicity conditions.

To address the test space challenge, WITCHER infers a set of *likely-ordering/atomicity conditions* by analyzing program data/control dependencies among NVM accesses. WITCHER then tests only those NVM states that violate the likely-ordering/atomicity conditions, significantly reducing the NVM state test space.

To mitigate the test oracle problem, WITCHER employs *output equivalence checking* between program executions with and without a (simulated) crash. NVM programs are often designed to provide atomic (all or nothing) semantics upon a crash at the operation granularity (*e.g.*, `insert`, `delete`), more formally, durable linearizability [64]. If an NVM program resuming from an NVM state that violates a likely-ordering/atomicity condition produces an output that is different from the executions without a crash, then we can confidently conclude that the program is not crash-consistent, and the violation is indeed a true crash consistency bug.

In this section, we introduce how WITCHER infers likely-ordering/atomicity conditions (§4.1.1) and performs output equivalence checking (§4.1.2) to validate the NVM states violating them.

4.1.1 Inference of Likely-ordering/atomicity Conditions

We propose a novel approach that analyzes program data/control dependencies among NVM accesses to infer likely-ordering/atomicity conditions enforcing persistence ordering and persistence atomicity guarantees among NVM accesses. Our key observation is that programmers often left some hints on what they want to ensure in the source code in the form of data/control dependencies, and we can infer the corresponding likely-ordering/atomicity con-

ditions. Witcher tests only NVM states that violate the inferred likely-ordering/atomicity conditions. In this way, Witcher uses likely-ordering/atomicity conditions to reduce NVM state test space without manual annotations. Witcher does not require prior knowledge of truth and does not assume the conditions are always correct; if two conditions contradict, we test both cases to discern which one is correct using output equivalence checking. In section §5.2.2, we present more generalized rules to infer likely-ordering/atomicity conditions beyond guarded protection.

4.1.2 Validation with Output Equivalence Checking

Witcher uses *output equivalence checking* to validate if an NVM state that violates an inferred likely-ordering/atomicity condition is indeed inconsistent, indicating a crash consistency bug. Many NVM programs, such as persistent key-value stores and indexes, aim to provide durable linearizability [64] at the operation granularity (*e.g.*, `insert`, `delete`). That is, upon a crash, an NVM program should behave as if the operation where the crash occurred is either fully executed or not at all executed (*i.e.*, all or nothing semantics). Therefore, Witcher can validate crash consistency by comparing the outputs of executions with and without a crash. If a program that recovers from an NVM state violating a likely-ordering/atomicity condition produces an output different from the executions without a crash, then we can confidently conclude that the program is not crash-consistent. If so, the violation of a likely-ordering/atomicity condition is a true bug.

Output equivalence checking allows Witcher to automatically detect persistence correctness bugs without manual annotations or a user-provided full consistency checker. Output equivalence checking requires that the test case is deterministic; *i.e.*, given the same input, a program should produce the same output. Moreover, output equivalence checking relies on

test cases, and thus some crash consistency bugs may not be detected if they do not produce visible symptoms (*e.g.*, segmentation-fault, different output, etc.) on the given test cases. This implies that we may have false negatives. However, any detected output divergence is indeed an indicator of a true correctness bug; *i.e.*, we do not have false positives.

4.2 Likely-linearization Point Inference

We propose DURINN, which detects persistence correctness bugs by inferring likely-linearization points.

DURINN is based on the novel observation on the gap between a linearization point, where the changes to a data structure becomes visible, and a 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 linearization and durability points, we derive three durable linearizability (DL) bug patterns that render a linearizable data structure not durably linearizable.

To tame the huge test space, DURINN uses two novel techniques: *adversarial crash state and thread interleaving construction* and *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 distance between a constructed crash state and a consistent state preserving DL conditions, thus increasing the likelihood of revealing persistence correctness 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.

In this section, we introduce the gap analysis between a linearization point and a durability point (§4.2.1), adversarial NVM state and thread interleaving construction (§4.2.2), and likely-linearization point inference (§4.2.3).

4.2.1 The Gap between Linearization Point and Durability Point

As illustrated in Figure 4.2, 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.

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 three DL bug patterns.

The first *Incompletely-Durable* bug pattern considers a crash *after DP* (region `R3` in Figure 4.2). 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. The second *Unrecovered-Durable* bug pattern considers a crash *before DP* (regions `R1` and `R2` in Figure 4.2). As a crash hap-

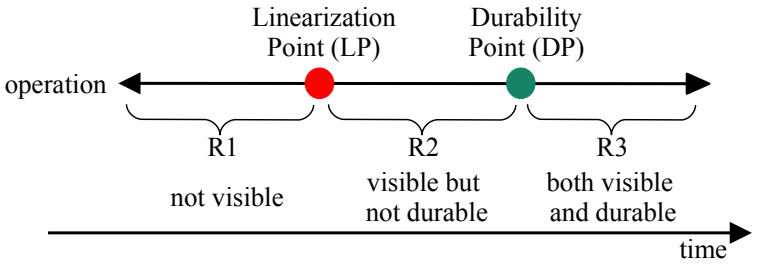


Figure 4.2: Linearization point and durability point split an operation into three-time intervals as per its visibility and durability guarantees.

pens before DP, any temporal change made by the crashed operation should not be visible after the resumption. The last *Visible-But-Not-Durable* bug pattern considers a crash *between LP and DP* (region `R2` in Figure 4.2). 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. In section §6.1, we present a detailed discussion about the three durable linearizability bug patterns, along with real-world examples detected by DURINN.

4.2.2 Adversarial NVM State and Thread Interleaving Construction

We propose an adversarial technique to effectively explore the huge test space in finding persistence correctness 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. To the best of our knowledge, DURINN is the first persistence bug detector using an adversarial testing approach.

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

4.2.3 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) NVM programs usually use guarded-protection [45] to ensure persistence atomicity; (3) concurrent programs usually make a memory region visible to other threads after initialing the memory region. 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 persistence bug detector that statically infers linearization points from concurrent NVM programs.

4.3 Program Trace Analysis

We use a *trace-based* approach to detect persistence performance bugs. Unlike finding persistence correctness bugs, which requires searching possible crashed NVM states, detecting persistence performance bugs does not need crash simulation and only requires tracking the NVM persistence state in program order. For example, to detect an extra flush performance bug, the persistence state of the cacheline to be flushed before executing the flush instruction is needed. We leverage the collected dynamic program trace and detect persistence performance bugs during NVM persistence simulation.

Chapter 5

Inferring Likely-ordering/atomicity Conditions for Persistence Correctness Bug Detection

This chapter introduces Witcher, which infers likely-ordering/atomicity conditions to detect persistence bugs from NVM programs. Witcher makes the following contributions:

- We propose a new NVM software testing approach that infers likely-ordering/atomicity conditions to effectively explore NVM state test space, and performs output equivalence checking to identify an incorrect execution without user-provided test oracles. To the best of our knowledge, Witcher is the first NVM testing tool that uses program-agnostic rules to find persistence correctness bugs from log-free NVM programs.

- Witcher detects 47 (36 new) persistence correctness bugs in NVM-backed key-value stores and PMDK library. Witcher does not suffer from test space explosion nor requires manual test oracles to detect them. The current Witcher prototype focuses on testing key-value stores in which operation interfaces are well known and thus output equivalence checking can be automatically performed. The proposed ideas can be extended and applied to other NVM programs beyond key-value stores.

In the following sections of this chapter, we first present the persistence correctness bugs (§5.1) with real-world code examples, then present the detailed design of Witcher (§5.2).

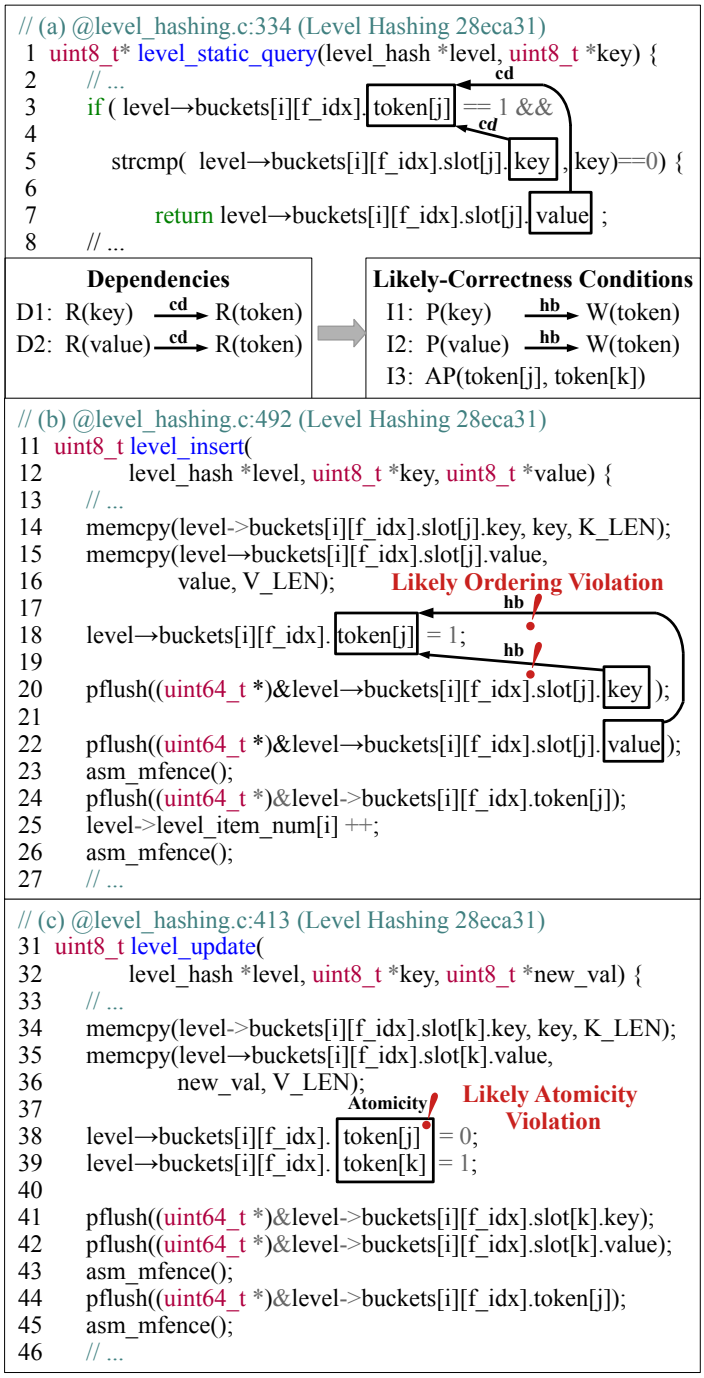
5.1 Persistence Correctness Bugs

As a processor can evict cache lines in an arbitrary order and does not support atomic update of multiple NVM locations, crash consistency is the problem of guaranteeing persistence ordering and atomicity of NVM locations according to program's semantics. Thus, violating these is the primary reason for correctness bugs in NVM programs. Based on the root causes, persistence correctness bugs are classified into two categories, *i.e.*, persistence ordering bug (§5.1.1) and persistence atomicity bug (§5.1.2).

5.1.1 Persistence Ordering Bugs

We found that many crash consistency and recovery mechanisms rely on a certain *persistence ordering* of NVM variables. However, a buggy NVM program may not maintain proper persistence ordering using a cache line flush and a store fence instructions when updating multiple NVM locations.

For example, Level Hashing [134] introduces *log-free* write operations. It maintains a flag token for each key-value slot where the token denotes if the corresponding key-value slot is empty or not. Figure 5.1(b) shows the log-free `level_insert` function. It intends to update the key-value slot (Lines 14, 15) before updating the token (Line 18). However, if a crash happens after the token's cache line is evicted (thus persisted) but before the key-value slot's cache line is not (before enforcing cache line flushes at Lines 20–22), an inconsistent state could occur – the token indicates that the corresponding key-value slot is non-empty, but the slot is never written to NVM. Thus, the garbage value can be read (as in Figure 5.3(h)), implying that the insert operation failed to provide an atomic (all or nothing) semantic upon a crash. The persistent barriers at Lines 20-23 should be moved before updating the token at Line 18.



`R(X)`: read `X` `W(X)`: write `X` `P(X)`: persist `X` `AP(X,Y)`: `X`, `Y` persisted atomically
`E1` $\xrightarrow{\text{cd}}$ `E2`: `E1` is control dependent on `E2` `E1` $\xrightarrow{\text{hb}}$ `E2`: `E1` should happen before `E2`

Figure 5.1: Using the likely-ordering/atomicity conditions inferred from (a), WITCHER finds two persistence correctness bugs (b) and (c) in Level Hashing.

5.1.2 Persistence Atomicity Bugs

To ensure the integrity of NVM data, many NVM programs rely on atomic update of NVM variables. However, a buggy NVM program may not correctly enforce *persistence atomicity* among multiple NVM updates. If a program crashes in the middle of a sequence of NVM updates, an inconsistent state may occur.

Figure 5.1(c) shows a persistence atomicity bug found in Level Hashing's `level_update` function. Level Hashing opportunistically performs a log-free update. If there is an empty slot in the bucket storing the old key-value slot, a new slot is stored to the empty slot (Lines 34, 35), and then the old and new tokens are modified (Lines 38, 39). Since the new slot is not overwritten to the old slot, Level Hashing can avoid costly logging operations. However, the code incorrectly assumes that updating two tokens is atomic. If a crash happens right after turning off the old token (Line 38) and the cache line of the old token is evicted (persisted), the crash consistency problem happens. Since the old token is persisted with 0 (empty) but the new token (Line 39) is not turned on, we permanently lose the updating key. To solve this bug, we have to persist two tokens atomically.

5.2 Design of Witcher

Figure 5.2 illustrates Witcher architecture that takes as input a target program (NVM-based persistent key-value stores) and a test case (some sequences of insert, delete, query, etc. operations); and reports as output detected correctness and performance bugs in the program. Witcher first instruments the program and runs the test case to collect a memory trace (§5.2.1). For correctness bugs, Witcher infers likely-ordering/atomicity conditions from the trace (§5.2.2), constructs a set of crash NVM images violating the likely-

ordering/atomicity conditions (§5.2.3), and performs output equivalence checking to validate if a likely-ordering/atomicity condition violation is a true persistence correctness bug (§5.2.4).

Witcher supports testing not only applications (key-value stores) but also PMDK libraries (*e.g.*, persistent heap management, transaction undo logging) as the PMDK libraries internally use low-level persistence primitives (such as `flush` and `fence` instructions) for crash consistency. Witcher provides limited support for multi-threading, and this section assumes testing single-threaded programs.

5.2.1 Tracing Memory Accesses

Witcher instruments an NVM program using an LLVM compiler pass [6] and executes the instrumented binary with a test case to collect the execution trace. We trace load, store/non-temporal store [1] (including the updated value), branch, call/return, flush and memory fence instructions.

Suppose we trace Level Hashing in Figure 5.1 using the test case with four operations in Figure 5.3(a). Figure 5.3(b) shows the trace of the last `level_static_query(k)`. Each trace includes a unique Trace ID (TID), a Static instruction ID (SID), which is the instruction location in the binary, and the instruction type. For `load` and `store`, Witcher additionally traces its address, length (not shown), and data (for `store`), and whether it accesses DRAM (white) or NVM (gray).

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

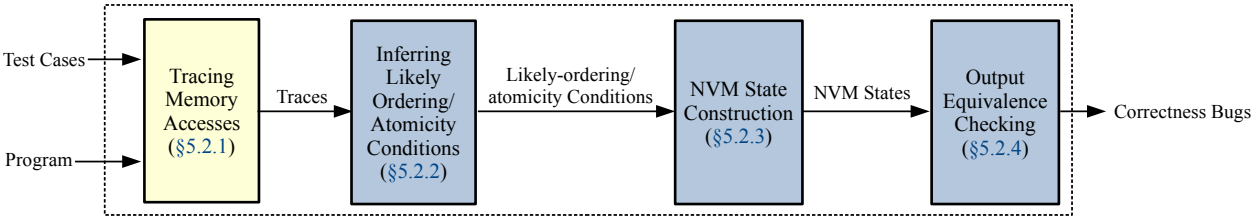


Figure 5.2: The architecture of WITCHER.

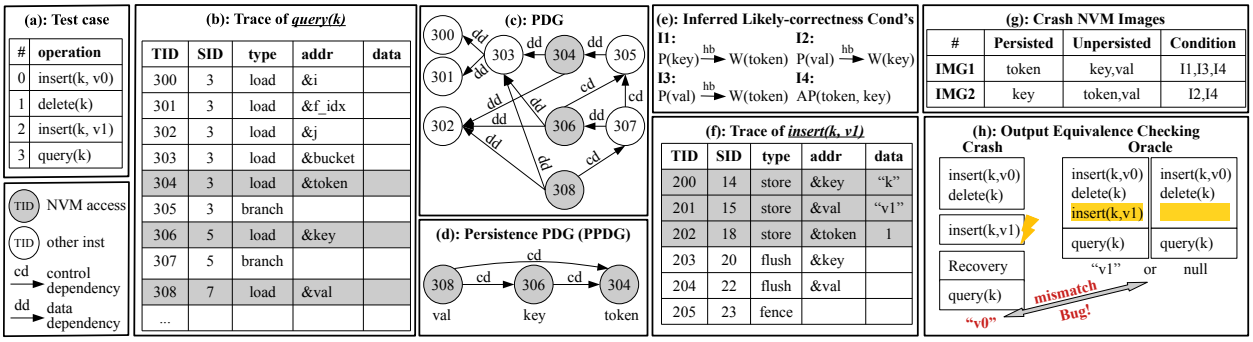


Figure 5.3: An example of WITCHER's correctness bug detection steps.

5.2.2 Inferring Likely-correctness Conditions: Likely-ordering/atomicity Conditions

WITCHER correlates program data/control dependencies with NVM crash consistency correctness conditions. We first demonstrate how we can infer likely-ordering/atomicity conditions using the Level Hashing example in Figure 5.1. We then describe a set of inference rules for likely persistence ordering conditions and likely persistence atomicity conditions. Lastly, we explain how WITCHER uses program dependence analysis to infer the likely-ordering/atomicity conditions from the trace.

Likely-ordering/atomicity Condition Inference Demonstration

Using the aforementioned Level Hashing example, let us demonstrate how we can infer a likely-ordering/atomicity condition from the query function `level_static_query` in Fig-

ure 5.1(a), and apply it to find the correctness bugs in `level_insert` and `level_update` in Figure 5.1(b) and (c). `level_static_query` reads the key/value only if the token is non-empty. In other words, there is *control dependency* between the read of a token and a key-value pair (Lines 3-7); *e.g.*, we denote it as `R(slot[j].key)` $\xrightarrow{\text{cd}}$ `R(token[j])`. We analyze the implication of this control dependency as follows.

We first refer to the common NVM programming pattern that uses a flag (`token`) to ensure the persistence atomicity of data (`key/value`) as *guarded protection*. We have observed this guarded protection pattern in many NVM programs including key-value stores [10, 92, 124], logging implementations [17, 53, 54, 63, 76, 110, 122], persistent data structures [27, 31, 55, 73, 79, 80, 97, 102], memory allocators [19, 33, 57, 103, 113], and file systems [29, 36, 37, 67, 125]. The guarded protection follows the following reader-writer pattern around a flag variable, which we call *"guardian"*; (1) The writer ensures that both key and value are "persisted before" the flag is persisted (Figure 5.1(b)); (2) The reader checks if the flag is set before reading the key and value, which we call *"guarded read"* (Figure 5.1(a)). The persistence ordering (for the writer side) and the guarded read (for the reader side) together ensure that the reader reads atomic (both old or both new) states of key and value.

From the guarded read pattern in Figure 5.1(a), we infer the first *likely-ordering condition*; a key-value pair should be persisted before a token – we denote it as `P(slot[j].key/value)` $\xrightarrow{\text{hb}}$ `W(token[j])`. We then extend it to the second *likely-atomicity condition* – the updates of two or more guardians should be atomic (*i.e.*, `AP(token[j]` and `token[k])`). Otherwise, an atomic update of multiple key-value slots cannot be guaranteed.

Later we find that `level_insert` violates the persistence ordering condition at Line 18, and `level_update` violates the persistence atomicity condition at Line 39. Witcher tests only NVM states that violate the inferred likely-ordering/atomicity conditions. For example, in `level_insert` we test only one case that a token is persisted but a key-value pair is

not persisted, which violates the writer pattern in the guarded protection. Similarly, in `level_update` we test two cases that one token is persisted and another token is not.

In this way, Witcher uses likely-ordering/atomicity conditions to reduce NVM state testing space without manual annotations. Witcher does not require prior knowledge of truth and does not assume the conditions are always correct; if two conditions contradict, we test both cases to discern which one is correct using output equivalence checking.

Inference Rules

Table 5.1 summarizes the inference rules. At a high level, each rule looks for control and/or data dependency *Hints* between NVM locations `X` and `Y` in a program. Witcher then infers a Persistence Ordering (PO) likely-correctness condition that "`X` should be persisted before `Y`" or a Persistence Atomicity (PA) condition that "`X` and `Y` should be persisted atomically". Witcher later constructs an NVM state that violates a likely-ordering/atomicity condition – *e.g.*, "`Y` is persisted, but `X` is not" (§5.2.3) and tests if the likely ordering/atomicity violation is a true crash consistency bug using output equivalence checking (§5.2.4). In other words, for two NVM addresses `X` and `Y`, if Witcher does not detect any dependency, it does not test such cases involving `X` and `Y`. Hence, it saves the test time, assuming that independent NVM objects do not lead to an inconsistent state.

(PO1) A data dependency implies a persistence ordering. Consider the code "`Y=X+1`" where the write of `Y` is data-dependent on the read of `X` (which we denote $\texttt{W(Y)} \xrightarrow{\text{dd}} \texttt{R(X)}$). From the data dependency, we infer a PO condition that for another code region where `X` and `Y` are updated, the developer would want `X` to be *persisted before* updating `Y` (*i.e.*, $\texttt{P(X)} \xrightarrow{\text{hb}} \texttt{W(Y)}$ where $\xrightarrow{\text{hb}}$ stands for happens-before). Otherwise, she may update `Y` based on "unpersisted" `X`, leading to an inconsistent state. Based on the reasoning, `PO1` in Table 5.1 says: for

#	Hint		Likely-correctness Cond		NVM Image	
	Example	Rule	Example	Rule	$\mathbb{P}$	$\mathbb{U}$
PO1	`Y=X+3;`	`W(Y)`$\xrightarrow{\text{dd}}$`R(X)`	`X=...;Y=...;`	`P(X)`$\xrightarrow{\text{hb}}$`W(Y)`	Y	X
PO2	`if(X){Y=3;}`	`W(Y)`$\xrightarrow{\text{cd}}$`R(X)`	`X=...;Y=...;`	`P(X)`$\xrightarrow{\text{hb}}$`W(Y)`	Y	X
PO3	`if(X){Z=Y+3;}`	`R(Y)`$\xrightarrow{\text{cd}}$`R(X)`	`Y=...;X=...;`	`P(Y)`$\xrightarrow{\text{hb}}$`W(X)`	X	Y
PA1	`if(X){M=N+3;}`	`R(N)`$\xrightarrow{\text{cd}}$`R(X)`	`X=...;Y=...;`	`AP(X,Y)`	X	Y
	`if(Y){K=J+3;}`	`R(J)`$\xrightarrow{\text{cd}}$`R(Y)`			Y	X

`R(X)`: read `X` `W(X)`: write `X` `P(X)`: persist `X` $\mathbb{P}$: persisted $\mathbb{U}$: unpersisted
`E1` $\xrightarrow{\text{cd}}$ `E2`: `E1` is control dependent on `E2` `E1` $\xrightarrow{\text{dd}}$ `E2`: `E1` is data dependent on `E2`
`E1` $\xrightarrow{\text{hb}}$ `E2`: `E1` should happen before `E2` `AP(X,Y)`: `X` and `Y` persisted atomically

Table 5.1: The inference rules `PO1`–`PO3` are for persistence ordering likely-correctness conditions and `PA1` is for persistence atomicity.

two NVM locations `X` and `Y`, if we find a Hint `W(Y)`$\xrightarrow{\text{dd}}$`R(X)`, we infer a likely PO condition `P(X)`$\xrightarrow{\text{hb}}$`W(Y)`. We later test an NVM state violating the PO condition where `Y` is persisted, but `X` is not.

(PO2) A control dependency implies a persistence ordering. Based on the same rationale, we infer another PO condition from the control dependency as well: *e.g.*, "`if(X) Y=1`". More formally, `PO2` says: for two NVM locations `X` and `Y`, if we find a Hint `W(Y)`$\xrightarrow{\text{cd}}$`R(X)`, we infer a likely PO condition `P(X)`$\xrightarrow{\text{hb}}$`W(Y)`. Then we test a state violating the PO condition where only `Y` is persisted.

(PO3) A guarded read implies a persistence ordering. As discussed in the demonstration, guarded protection is a common NVM programming pattern. It achieves the atomicity of data using the writer-side persistence ordering and the reader-side guarded read. Based on this observation, if we see a guarded read pattern at a reader side, we infer a PO condition at a writer side. In other words, `PO3` says: for two NVM locations `X` and `Y`, if we find a Hint `R(Y)`$\xrightarrow{\text{cd}}$`R(X)`, we infer a Likely PO Condition `P(Y)`$\xrightarrow{\text{hb}}$`W(X)`. We note that here `X` is a guardian in the guarded read pattern (*e.g.*, `token` in Figure 5.1) and thus it should be persisted last (after `key` and `value`). We then validate an NVM state violating the condition such that `X` is persisted but `Y` is not.

(PA1) Guardian implies persistence atomicity. As in the `PO3` likely-correctness condition, we can find a set of guardians: *e.g.*, `token[j]` and `token[k]` in Figure 5.1. A program state could be inconsistent if all the guardians are not updated atomically — no one guards the guardians. Based on this observation, we infer a PA likely-correctness condition such that two or more guardians should be atomically updated. `PA1` says: for two guardians `X` and `Y` from `PO3`, we infer the Likely PA Condition `AP(X,Y)` that `X` and `Y` should be atomically persisted. We later test NVM states such that only one guardian is persisted. This approach allows us to reduce testing space significantly because we will not test persistence atomicity for well-guarded NVM data. For example, if a program applies the guarded read patterns on `key` and `value` in all places (using `token` as a guardian), then we do not test persistence atomicity between them. Given N guardians, there will be N^2 `PA1` conditions. To avoid scalability issues, when checking a `PA1` violation, Witcher keeps track of a set of N guardians instead of N^2 conditions, and checks if two stores before a fence belong to the set.

Data/Control Dependence Analysis

Witcher performs program dependence analysis to infer likely-ordering/atomicity conditions from the source codes and execution traces. Witcher first constructs a Program Dependence Graph (PDG) [40, 49, 101] where a node represents a traced instruction, and an edge represents data or control dependency. Then, Witcher simplifies the PDG into what we called Persistence Program Dependence Graph (PPDG) that captures dependencies between NVM accesses to make it easy to apply the likely-ordering/atomicity condition inference rules. For example, Figure 5.3(c) shows the PDG of the trace (b), and (d) shows the PPDG. Witcher uses a mix of static and dynamic trace analysis to construct a PDG. When instrumenting the source code for tracing (§5.2.1), it performs static analysis to capture register-level data and control dependency. Then it extracts memory-level data dependence by analyzing

memory-level data-flow in the collected trace. This dynamic memory-level data dependency analysis improves PDG's precision compared to static-only analysis, which suffers from the imprecision of pointer analysis. The static instruction IDs (binary address) are used to map static and dynamic information.

Witcher converts a PDG to a PPDG as follows. Initially, the PPDG has only (gray) NVM nodes. Witcher traverses the PDG from one NVM node to another NVM node. If there is at least one control-flow edge along the path, it adds a control-flow edge in the PPDG. If a path includes only data-flow edges, it adds a data-flow edge in the PPDG. No path implies no dependency.

Given the PPDG, Witcher then applies the inference rules in Table 5.1. For each edge and two nodes in the PPDG, Witcher considers the type of edge (control vs. data) and the type of instructions (`store` vs. `load`). When Witcher finds a Hint, it records the corresponding Likely-ordering/atomicity Condition. For example, the PPDG in Figure 5.3(d) shows that `R(key)` $\xrightarrow{\text{cd}}$ `R(token)`. Based on `PO3`, we infer the PO condition `I1: P(key)` $\xrightarrow{\text{hb}}$ `W(token)` in (e). Similarly, we can infer the PO conditions `I2` and `I3`. Moreover, as `token` and `key` are guardians for guarded reads, based on `PA1`, we infer the PA condition `I4: AP(token,key)`.

5.2.3 NVM State Construction

The next step is to generate a set of crash NVM images[2] that violate the likely-ordering/atomicity conditions. Later in §5.2.4, we will describe how Witcher loads these NVM images and uses output equivalence checking to validate if a likely-ordering/atomicity condition violation is a true bug or not.

At a high level, Witcher generates crash NVM images as follows. Witcher takes as input

[2]In PMDK, an NVM image is a regular file containing an NVM heap state created, loaded, and closed by PMDK APIs [59].

the same trace used to collect likely-ordering/atomicity conditions and performs cache and NVM simulations along the trace. During the simulation, Witcher cross-checks if there is any violation of likely-ordering/atomicity conditions. Each violating NVM state forms a crash NVM image to test. Witcher produces a set of crash NVM images for further validation.

Simulating Cache and NVM States

The goal of the cache/NVM simulations is to generate only feasible NVM states that violate likely-ordering/atomicity conditions but still obey the semantics of a persistence control at a cache line granularity (*e.g.*, the effects of a `flush` instruction). Starting from the empty cache and NVM states, Witcher simulates the effects of `store`, `flush`, and `fence` instructions along the trace while honoring the memory (consistency) model of a processor. In particular, Witcher supports Intel's x86-64 architecture model, as in Yat [78]. The following two rules are, in particular, relevant to the cache/NVM simulations: (1) A `fence` instruction guarantees that all the prior `flush`-ed stores are persisted. (2) A processor does not reorder two store instructions in the same cache line (following the x86-TSO memory consistency model [62, 116]).

Consider the trace of Level Hashing's `level_insert` code in Figure 5.3(f). After simulating the first three `store` instructions (TID 200-202), there could be multiple valid cache/NVM states. For example, the data ``k'' for `key` could either remain in a cache (unpersisted) or could be evicted (persisted). The same is true for the `val` and `token`. However, after finishing the execution of the last `fence` instruction (TID 205), `key` and `val` are guaranteed to be persisted (due to `flush` and `fence`). Still, `token` could be either unpersisted or persisted.

Detecting Likely-ordering/atomicity Condition Violations

During the simulation, Witcher checks if there could be an NVM state that violates a likely-ordering/atomicity condition before executing each `fence` instruction because the `fence` ensures a persistent state change. Witcher considers all possible persisted/unpersisted states while honoring the above cache/NVM simulation rules.

Consider the trace of Level Hashing's `level_insert` code in Figure 5.3(f) again. Before we execute the last `fence` instruction (TID 205), we check the four likely-ordering/atomicity conditions against the trace as shown in (e). For instance, `I1` says that `P(key)` $\xrightarrow{\text{hb}}$ `W(token)`. The state violating the PO condition is the one that `token` is persisted, but `key` is not. We check if this PO violation is feasible in this code region (before the `fence`). The answer is yes – a program crashes between the TID 202 `store` and the TID 203 `flush` instructions, and the cache line for `token` is evicted (persisted) but not for `key` and `val` (unpersisted). This forms the first crash NVM image `IMG1` in (g). Similarly, we can find `IMG1` is also the state that `I3` and `I4` are violated. We can also find the second `IMG2` in (g) violating `I2` and `I4`.

Each crash NVM image is indeed represented as a pair of a fence ID and a set of store IDs, which specifies where to crash and which stores to be persisted, respectively. Witcher repeats the process along the trace and generates a set of crash NVM images that will be validated in the next step.

5.2.4 Durable Linearizability Validation

Witcher validates the crash NVM images violating likely-ordering/atomicity conditions and detects crash consistency bugs using output equivalence checking. In particular, Witcher focuses on testing durable linearizability [64]. That is, a crash-consistent NVM program should behave as if the operation where the crash occurred is either fully executed (committed) or

not at all executed (rolled back). Thus, the program resumed from a crash NVM image should produce the same output as one of these two committed or rolled-back executions, which we call *oracles*.

Consider the example in Figure 5.3 again. Using the test case `insert(k,v0)`, `delete(k,v0)`, `insert(k,v1)`, and `query(k)` in (a), we analyzed the trace of the third `insert(k,v1)` operation in (f) and generated two crash NVM images in (g). The first `IMG1` reflects an NVM state that the first two operations, `insert(k,v0)` and `delete(k,v0)`, are correctly performed, and the program crashes in the middle of the third `insert(k,v1)` where only `token` is persisted, and `key` and `value` remain unpersisted – *i.e.*, `IMG1` has the old value `v0`.

Witcher generates two oracles to compare. The first oracle reflects an execution where the crashed operation is committed – thus we run the test case `insert(k,v0)`, `delete(k,v0)`, `insert(k,v1)`, and `query(k)` (no crash) and records `v1` (the new value) as the output of `query(k)`. The second oracle mimics an execution where the crashed operation is rolled back – we run the same test case without the third `insert(k,v1)` and log `null` as the output of `query(k)`. Altogether, the oracles say that the correct output of the last `query(k)` is either `v1` or `null`.

Witcher uses the same test cases (used for tracing and inference) for output equivalence checking. Witcher loads a crash NVM image, runs a recovery code (if it exists), executes the rest of the test cases, records their outputs, and compares them with the oracles. For example with `IMG1`, `query(k)` returns the old value `v0` (as neither the deletion of `k` nor the insertion of new value `v1` was persisted) – Witcher detects the mismatch and reports the test case and the crash NVM image information (the crash location as the `fence` TID, and the persistence state as the persisted `store` ID). On the other hand, a similar analysis with the second `IMG2` shows that the output (`null`) matches the oracles, so Witcher does not report it as correctness bugs.

One key benefit of output equivalence checking is that all the reported cases indeed indicate buggy inconsistent states (no false positives). Nonetheless, many cases may share the same root cause: *e.g.*, a bug in `insert` operation may repeatedly appear in a trace if the test case has many `insert` calls. To help programmers analyze the root causes, Witcher clusters the bug reports according to operation type (*e.g.*, `insert`, `delete`) and execution path (a sequence of basic blocks) that appeared in the trace. We found that our clustering scheme significantly facilitates the root cause analysis. After one root cause is found, reasoning about the redundant cases along the same program path is relatively simpler. Multiple clusters may share the same root cause.

Chapter 6

Inferring Likely-linearization Points for Persistence Correctness Bug Detection

This chapter introduces DURINN, which infers likely-linearization points to detect persistence correctness bugs from NVM programs. DURINN makes the following scientific contributions:

- We present three durability linearizability bug patterns after performing the 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 persistence correctness bugs in an active and scalable manner.
- DURINN reports 27 (15 new) persistence correctness bugs and outperforms state-of-the-art NVM testing tools in terms of bug detection effectiveness and test space reduction.

In the following sections of this chapter, we first present the three durability linearizability bug patterns (§6.1), then introduce the detailed design of DURINN (§6.2).

6.1 Durable Linearizability Bugs

This section presents three durable linearizability bug patterns derived from the gap analysis, along with real-world examples detected by Durinn (§6.1.1 - §6.1.3).

6.1.1 DL Bug Pattern 1: An Incompletely-Durable Bug

The first *Incompletely-Durable* bug pattern considers a crash *after DP* (region `R3` in Figure 4.2). 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 6.1 illustrates the Incompletely-Durable bug pattern. Since the crash happens after DP of `T1`'s `insert(K,V)`, to be 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 6.4(a) shows a part of rehashing code in P-CLHT [80], 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.

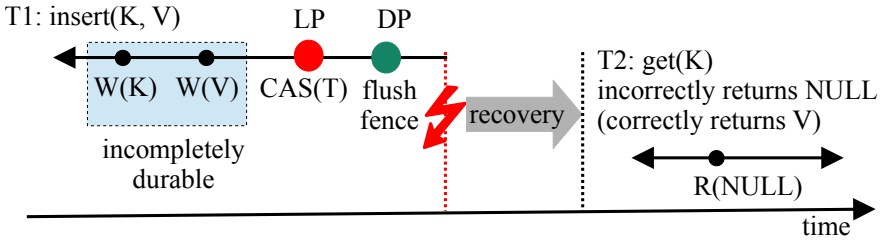


Figure 6.1: An *Incompletely-Durable* bug pattern. If a crash occurs after DP, the `insert` operation should make all its effects durable completely. Otherwise, the `get` operation after a crash may not be able to see its effect, producing wrong output.

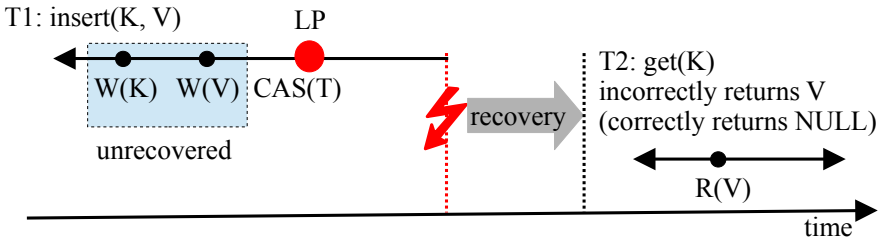


Figure 6.2: An *Unrecovered-Durable* bug pattern example. If a crash happens before DP, the `insert` operation should recover (undo) any partially durable effect. Otherwise, the `get` operation after a crash may see its partial effect, producing wrong output.

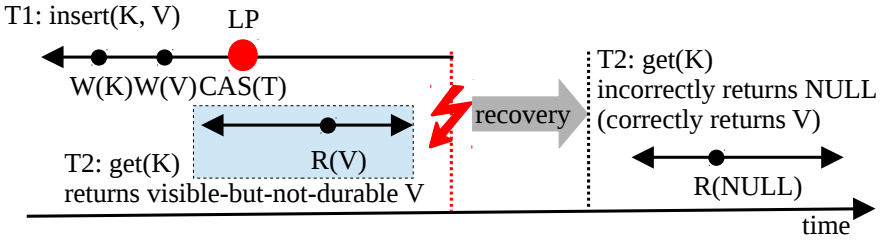
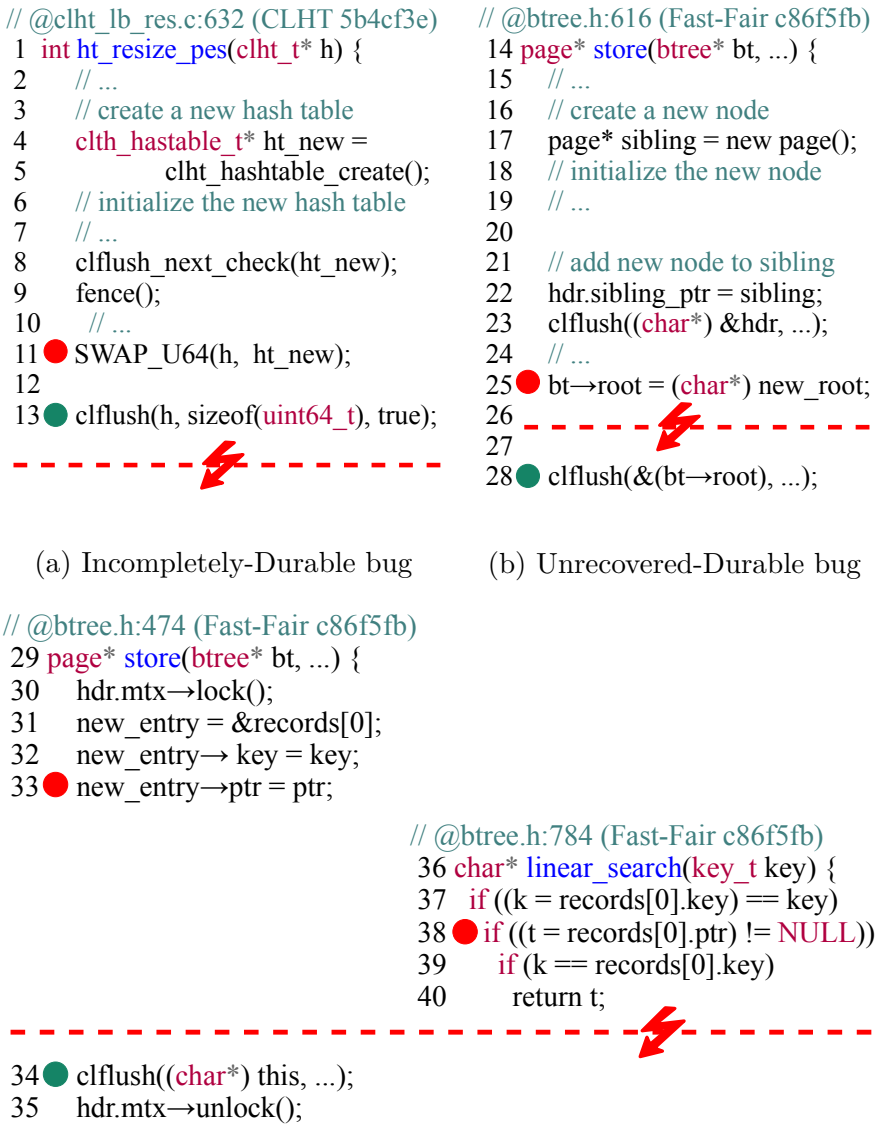


Figure 6.3: A *Visible-But-Not-Durable* bug pattern. For a crash between LP and DP of `insert`, the concurrent `get` may observe and return the visible-but-not-durable value. However, the second `get` after a crash may not be able to return the same value as the effect of `insert` is not durable.



(a) Incompletely-Durable bug

(b) Unrecovered-Durable bug

(c) Visible-But-Not-Durable bug

Figure 6.4: Durable linearizability bug examples in P-CLHT [80] (a) and Fast-Fair [55] (b) and (c). A red circle represents LP; green represents DP; and a red lightning bolt represents a crash.

6.1.2 DL Bug Pattern 2: An Unrecovered-Durable Bug

The second *Unrecovered-Durable* bug pattern considers a crash *before DP* (regions `R1` and `R2` in Figure 4.2). As a crash happens before DP, any temporal change made by the crashed op-

eration 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 6.2 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" [45] and we discuss it in detail in §6.2.2.

Figure 6.4(b) shows an *Unrecovered-Durable* bug from Fast-Fair [55], 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). 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.

6.1.3 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* (region `R2` in Figure 4.2). 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 6.3 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, one operation may help persist the update on a synchronization variable of another operation. The helping logic is analogous to the linearizability programming idiom in which one thread helps fix temporal inconsistency on behalf of another thread.

Figure 6.4(c) shows a *Visible-But-Not-Durable* bug from Fast-Fair [55]. 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.

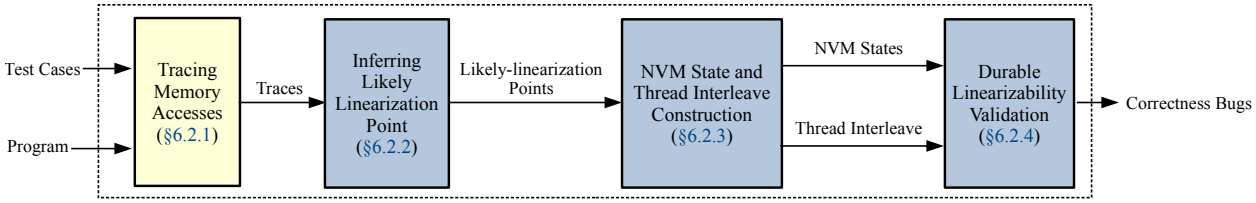


Figure 6.5: The overall architecture of DURINN.

6.2 Design of DURINN

We present the overall architecture of DURINN at Figure 6.5. 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 (§6.2.1). DURINN then infers likely-linearization points from the trace (§6.2.2). Given the memory trace and the identified likely-linearization points, DURINN performs adversarial NVM state and thread interleaving construction (§6.2.3) 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 (§6.2.4).

6.2.1 Tracing Memory Accesses

DURINN instruments all NVM memory accesses (load, store[1]) and NVM heap allocation. DURINN also traces control flow transfers (branch, function call) because our likely-linearization point inference (§6.2.2) 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 (§6.2.3), we instrument all flush and memory fence instructions. We also trace lock operations for adversarial thread interleaving construction (§6.2.3). For durable linearizability validation (§6.2.4), we trace the value of each store instruction.

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

We implement an LLVM compiler pass [6] 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.

6.2.2 Inferring Likely-correctness Conditions: Likely-Linearization Points

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 [45]. 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 6.6(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,

```
// 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 6.6: Examples for *Guarded-Protection* and *Publish-after-Initialization* from (a) CCEH [97] and (b) NVTraverse [43]. Likely-linearization points are at line 7 and 13 in (a) and (b), respectively.

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 6.6(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.

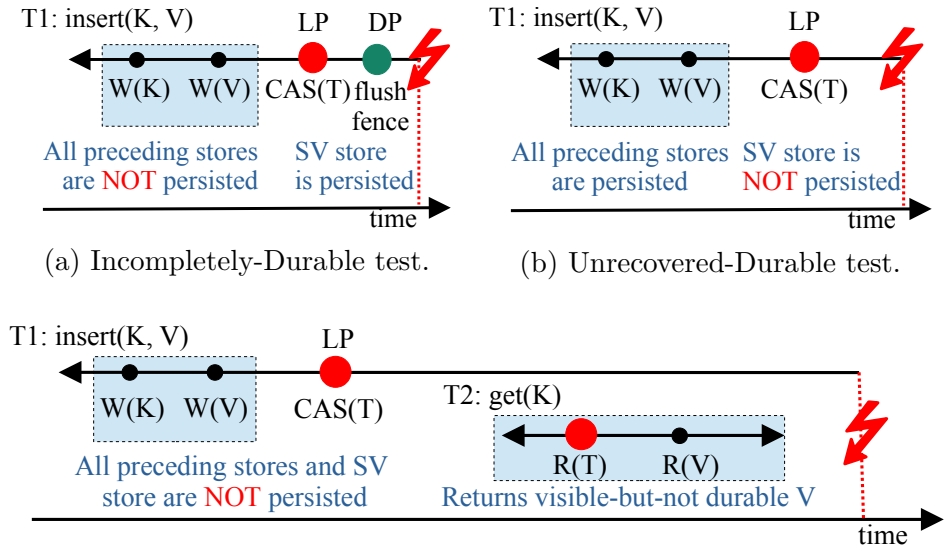
6.2.3 NVM State and Thread Interleave Construction

In this section, we first describe our adversarial construction approaches for each DL bug pattern. We then introduce our cache/NVM simulations to generate feasible NVM states for the validation.

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 6.1). 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 6.7(a) 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



(a) Incompletely-Durable test.

(b) Unrecovered-Durable test.

(c) Visible-But-Not-Durable test.

Figure 6.7: Adversarial test strategies for Incompletely-Durable, Unrecovered-Durable, and Visible-But-Not-Durable bugs. LP: linearization point. DP: durability point. SV: synchronization variable.

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).

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 6.2). 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 6.7(b) 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.

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 6.3). Unlike the prior two bug patterns, testing Visible-But-Not-Durable bugs should be performed in a context sensitive manner. Figure 6.7(c) 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 6.7(c), 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 6.7(c).

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.

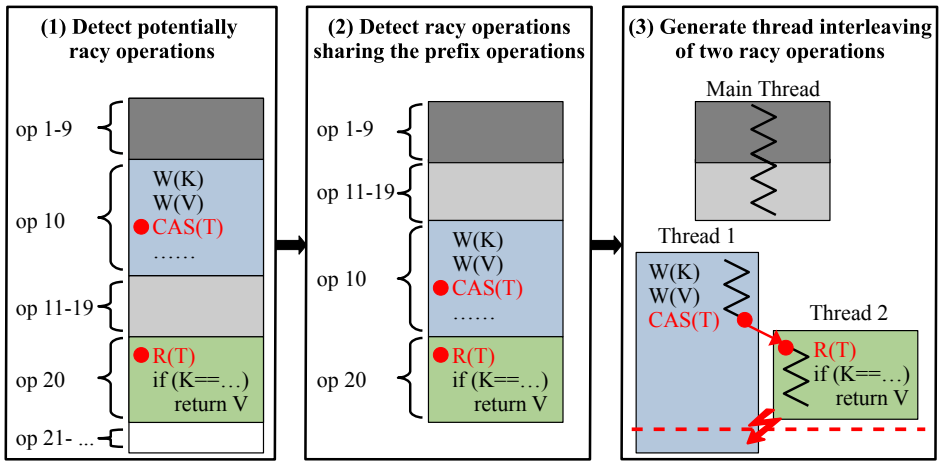


Figure 6.8: The workflow of adversarial NVM state and thread interleaving construction for Visible-But-Not-Durable bugs.

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 6.8 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.

(1) Finding racy operations. The first step is to find potentially race operations. The inputs for the analysis are a single-threaded execution trace (§6.2.1) and the inferred likely-linearization points (§6.2.2). 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 6.8 (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 6.8 (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 re-orders operations such that the two operations have the common prefix operations and places two potentially racy operations consecutively. In Figure 6.8 (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 6.8 (3), thread 1 writes synchronization variable `T` first, which is LP, then thread 2 preempts and reads `T` then return the `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 6.8). 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.

Cache and NVM simulations

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 [62, 116].

6.2.4 Durable Linearizability Validation

DURINN runs the NVM data structure under test from an NVM crash image (generated in §6.2.3) 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 §2.3); and whether the crashed operation is either fully executed or not at all executed (C3 condition).

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-value pairs 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. Durinn provide similar validating operations for other NVM data structures: *e.g.*, persistent array and queue.

Chapter 7

Analyzing Program Trace for Persistence Performance Bug Detection

Persistence performance bugs incurs unnecessary program performance degradation but do not cause an inconsistent state, yet it requires significant developers' time and effort to spot and fix them. Detecting persistence performance bugs have been well studied by previous works [84, 98]. In this dissertation, we use the similar persistence performance bug patterns from previous works to detect persistence performance bugs from our collected memory trace. We treat our trace-based persistence performance bug detection as a bonus of our proposed solutions. We do not claim the novelty of our trace-based persistence performance bug detection and do no claim it as a contribution of this dissertation.

We use a *trace-based* approach to detect persistence performance bugs. Unlike finding persistence correctness bugs, which requires searching possible crashed NVM states, detecting persistence performance bugs does not need crash simulation and only requires tracking the NVM persistence state in program order. We leverage the collected dynamic program trace and detect persistence performance bugs during NVM persistence simulation.

We detect the following performance bugs based on our trace-based cache/NVM simulation.

(1) Unpersisted performance bugs. Some NVM programs unnecessarily place volatile data that does not require persistence in NVM. Developers do not use flush/fence for volatile data. However, NVM accesses have higher latency than DRAM accesses. Developers should have placed them in DRAM. We report an unpersisted performance bug if a store still remains in the cache (not persisted) at the end of simulation yet it passes a durable linearizability validation.

(2) Extra flush performance bugs. An extra flush instruction on an NVM variable causes unnecessary high overhead. Removing the extra flush instructions does not break the correctness of an NVM program. When simulating a flush instruction, we report an extra flush performance bug if all prior stores have already been flushed by prior flush instructions.

(3) Extra fence performance bugs. An extra fence instruction causes unnecessary high overhead. Removing the extra fence instructions does not break the correctness of an NVM program. When simulating a fence instruction, we report an extra fence performance bug if there are no preceding flush instructions.

(4) Extra logging performance bugs. When an NVM program relies on a logging library (*e.g.*, Intel's PMDK) for crash consistency, the NVM data should be (undo) logged before it is modified the first time. Logging the same NVM region redundantly in a transaction is a performance bug. For transactional NVM programs, we report an extra logging performance bug if a memory region or its subset has already been logged by preceding logging operation in the same transaction.

Chapter 8

Implementation

We implemented our tracing and data flow analysis in LLVM [6]. We built our program dependency analysis based on Giri [112], a dynamic program slicing tool implemented in LLVM. 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. To support our durable linearizability validation, we provide a template driver with placeholders for test program initialization, recovery, and operations (*e.g.*, lookup/insert/delete). Note that users do not need to specify the correct output (*e.g.*, `E_NOTFOUND` v.s. `NULL`) because our proposed solutions check if the test and oracle executions produce the same outputs. We run like-correctness condition inference, NVM state and thread interleaving construction, and durable linearizability validation in parallel. 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 `mmap`-ed NVM heap are the same across different executions, we set `PMEM_MMAP_HINT` environment variable [61]. Witcher implementation comprises around 3,600 lines of C++ code and 4,400 lines of Python code. Durinn implementation comprises around 1,900 lines of C++ code and 2,700 lines of Python code. The Witcher prototype is available at https://github.com/cosmoss-vt/witcher.

Chapter 9

Evaluation

This section presents the evaluation of our proposed solutions. We first introduce the evaluation methodology (§9.1), then present the experiments of Witcher (§9.2) and Durinn (§9.3), and lastly present the results of our trace-based persistence performance bug detection (§9.4).

9.1 Evaluation Methodology

9.1.1 Tested NVM Programs

We evaluate Witcher and Durinn with five groups of NVM programs (Table 9.1). The first group includes five highly optimized persistent key-value indexes, which are the backbone of many key-value stores and storage systems. For high performance, they all have their own crash consistency mechanism using low-level (LL) persistence primitives such as `flush` and `fence` instructions. The second group includes seven concurrent persistent indexes converted by RECIPE [80]. We used three different versions/configurations of P-CLHT to compare with Agamotto [98]. Similar to the first group, they implement index-specific custom crash consistency logic using low-level primitives for performance (except for P-CLHT-Aga-TX using PMDK transaction). The third group includes five concurrent lock-free persistent indexes converted by NVTraverse [43]. Similar to the first and second groups, they implement index-specific custom crash consistency logic using low-level primitives for performance. The

	Application	Version	Lib	Design	Core NVM Construct	Concurrency	WITCHER Tested	DURINN Tested	Perf Bug Tested
NVM KV Index	WOART [79]	5b4cf3e	pmdk v1.8	LL	radix tree	ST	✓	×	✓
	WORT [79]	5b4cf3e	pmdk v1.8	LL	radix tree	ST	✓	×	✓
	Fast fair [55]	c86f5fb	pmdk v1.8	LL	b+ tree	LB	✓	✓	✓
	Level hash [134]	28eca31	pmdk v1.8	LL	hash table	ST	✓	×	✓
	CCEH [97]	d53b336	pmdk v1.8	LL	hash table	LB	✓	✓	✓
RECIPE	P-ART [80]	5b4cf3e	pmdk v1.8	LL	radix tree	LB	✓	✓	✓
	P-BwTree [80]	5b4cf3e	pmdk v1.8	LL	b+ tree	LF	✓	×	✓
	P-CLHt [80]	5b4cf3e	pmdk v1.8	LL	hash table	LB	✓	✓	✓
	P-CLHt-aga [80]	53923cf	pmdk v1.8	LL	hash table	LB	✓	×	✓
	P-CLHt-aga-tx [80]	53923cf	pmdk v1.8	TX	hash table	LB	✓	×	✓
	P-Hot [80]	5b4cf3e	pmdk v1.8	LL	trie	LB	✓	✓	✓
	P-Masstree [80]	5b4cf3e	pmdk v1.8	LL	b tree + trie	LB	✓	✓	✓
NVTraverse	P-LF-bst [43]	5fa1dee	pmdk v1.8	LL	binary search tree	LF	×	✓	×
	P-LF-hash [43]	5fa1dee	pmdk v1.8	LL	hash table	LF	×	✓	×
	P-LF-list [43]	5fa1dee	pmdk v1.8	LL	linked list	LF	×	✓	×
	P-LF-skiplist [43]	5fa1dee	pmdk v1.8	LL	skiplist	LF	×	✓	×
	P-LF-queue [42]	08fecfb	pmdk v1.8	LL	queue	LF	×	✓	×
PMDK	B-Tree	v1.4	pmdk v1.8	TX	b tree	ST	✓	×	✓
	C-Tree	v1.4	pmdk v1.8	TX	crit-bit tree	ST	✓	×	✓
	RB-Tree	v1.4	pmdk v1.8	TX	red-black tree	ST	✓	×	✓
	RB-Tree-aga	v0.4	pmdk v1.8	TX	red-black tree	ST	✓	×	✓
	Hashmap-tx	v1.4	pmdk v1.8	TX	hash table	ST	✓	×	✓
	Hashmap-atomic	v1.4	pmdk v1.8	LL	hash table	ST	✓	×	✓
	pmdk-array [7]	v1.8	pmdk v1.8	LL	array	LB	×	✓	×
	pmdk-queue [9]	v1.8	pmdk v1.8	TX	queue	LB	×	✓	×
Server	Memcached	8f121f6	PMDK v1.8	LL	hash table	LB	✓	×	✓
	Redis	v3.2	PMDK v1.8	TX	hash table	ST	✓	×	✓

LL: low-level persistence primitives **TX**: transaction
ST: single-threaded **LB**: lock-based **LF**: lock-free

Table 9.1: Tested NVM programs.

fourth group includes eight (example) persistent indexes in PMDK. They used PMDK's low-level (LL) or transactional (TX) persistence programming model. We used two versions of RB-tree for the comparison with Agamotto. The last group includes PMDK-based `Memcached` and `Redis` using PMDK's LL and TX persistence APIs, respectively.

All the tested NVM programs 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 programs use PMDK library (`libpmemobj`) for persistent memory allocation or transaction. For some programs that originally used a volatile memory allocator to emulate NVM using DRAM, we modified the code to use the PMDK memory allocator. We did not add or remove any persistence

primitives, nor introduce additional memory operations, which may potentially affect the bug detection evaluation. Witcher and Durinn trace and analyze both applications and PMDK libraries such as persistence heap allocation and transactional undo logging logic.

9.1.2 Test Cases

Witcher and Durinn both require a deterministic test case such that it produces the same output for a given input for durable linearizability validation. Any deterministic test case with good code coverage would suffice.

We randomly generate a list of operations, keys, and values as test cases for evaluation. For operation parameters, to make some dependent operations more meaningful, we assign a higher probability to (1) generate an unused key for `insert`; and (2) to generate a used key for the other operations – `delete`, `update`, `query`, and `scan` – which work on existing keys. We run the NVM programs with a test case consisting of 1,000 and 2,000 randomly generated operations for Witcher and Durinn evaluation, respectively. We found that 1,000 and 2,000 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.

9.1.3 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.

9.2 Detecting Persistence Correctness Bugs by Inferring Likely-ordering/atomicity Conditions

This section presents the following experiment results for WITCHER.

- We report and analyze the persistence correctness bugs (§9.2.1) detected by WITCHER along with detailed statistics (§9.2.2).
- We compare WITCHER with other NVM persistence bug detectors in terms of test space reduction (§9.2.3) and bug detection effectiveness (§9.2.4).
- We also evaluate WITCHER with non-key-value store NVM programs (§9.2.5).

9.2.1 Detected Persistence Correctness Bugs

WITCHER detected 47 (36 new) persistence correctness bugs from 18 programs. There were 25 persistence ordering bugs and 22 persistence atomicity bugs. All the bugs were confirmed by the developers. Table 9.2 presents the source code locations, impacts, and fix strategies of the detected correctness bugs.

The detected bugs have diverse impacts: lost, unexpected, duplicated key-value pairs; unexpected operation failure; and inconsistent structure. For example, a crash in the middle of rehashing operation in Level Hashing (Bug IDs 17 and 18 in Table 9.2) may lead to lost, unexpected, duplicated key-value pairs since the metadata is not consistent with the stored key-value pairs. In FAST-FAIR (Bug ID 5), if a crash happens while splitting the root node and right before setting the new root node, the B+tree will be in an illegal state: the root node connects to a sibling node. Any further operation on the B+tree will lead to a program crash.

Name (Total #Bugs)	Bug ID	New	Code	Type	Description	Impact	Fix strategy
libpmemobj (1)	1	✓	memblock.c:1337	C-O	Incorrect persistence order in allocation	Inconsistent structure	persistence reorder [3]
WOART (1)	2	✓	woart.c:727	C-A	Atomicity in node split	Inconsistent structure	inconsistency-recoverable design
FAST-FAIR (4)	3	✓	btree.h:224	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
	4	✓	btree.h:213	C-A	Partial inconsistency is never recovered	Inconsistent structure	inconsistency-recoverable design
	5	×	btree.h:576	C-A	Atomicity in node splitting	Inconsistent structure	logging/transaction
	6	×	btree.h:299	C-A	Atomicity in node merge	Inconsistent structure	logging/transaction
Level Hashing (17)	7	✓	level_hashing.c:492	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	8	✓	level_hashing.c:507	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	9	✓	level_hashing.c:417	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	10	✓	level_hashing.c:610	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	11	✓	level_hashing.c:616	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	12	✓	level_hashing.c:657	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	13	✓	level_hashing.c:677	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	14	✓	level_hashing.c:545	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	15	✓	level_hashing.c:560	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	16	✓	level_hashing.c:445	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [5]
	17	✓	level_hashing.c:112	C-A	Atomicity in rehashing	Inconsistent structure	logging/transaction
	18	✓	level_hashing.c:228	C-A	Atomicity in rehashing	Inconsistent structure	logging/transaction
	19	✓	level_hashing.c:609	C-A	Atomicity between two metadata	Duplicated key-value	inconsistency-tolerable design
	20	✓	level_hashing.c:665	C-A	Atomicity between two metadata	Duplicated key-value	inconsistency-tolerable design
	21	✓	level_hashing.c:685	C-A	Atomicity between two metadata	Duplicated key-value	inconsistency-tolerable design
	22	✓	level_hashing.c:416	C-A	Atomicity between two metadata	Lost key-value	merge to word size [5]
	23	✓	level_hashing.c:444	C-A	Atomicity between two metadata	Lost key-value	merge to word size [5]
CCEH (2)	24	×	CCEH_MSB.cpp:103	C-A	Atomicity in rehashing	Inconsistent structure	inconsistency-recoverable design
	25	✓	CCEH_MSB.cpp:29	C-A	Partial inconsistency is never recovered	Unexpected op failure	inconsistency-recoverable design
P-ART (2)	26	✓	N16.cpp:15	C-A	Atomicity between metadata and key-value	Inconsistent structure	inconsistency-tolerable design [13]
	27	✓	N4.cpp:17	C-A	Atomicity between metadata and key-value	Inconsistent structure	inconsistency-tolerable design [13]
P-BwTree (2)	28	✓	bwtree.h:2012	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives
	29	✓	bwtree.h:2369	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives
P-CLHT (1)	30	✓	clht_lb_res.c:166	C-O	Missing persistence primitives	Lost key-value	add persistence primitives [12]
P-CLHT-Aga (3)	31	✓	clht_lb_res.c:177	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
	32	✓	clht_lb_res.c:578	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
	33	×	clht_lb_res.c:583	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
P-CLHT-Aga-TX (2)	34	×	clht_lb_res.c:559	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
	35	×	clht_lb_res.c:583	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
P-HOT (3)	36	✓	TwoEntriesNode.hpp:30	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives [12]
	37	✓	HOTRowexNode.hpp:315	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives [12]
	38	✓	HOTRowex.hpp:270	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives [12]
P-Masstree (1)	39	✓	masstree.h:1378	C-A	Atomicity in node splitting	Inconsistent structure	logging/transaction
B-Tree (1)	40	×	btree_map.c:201	C-A	Missing logging in a transaction	Inconsistent structure	add logging
RB-Tree (1)	41	×	rbtree_map.c:417	C-A	Missing logging in a transaction	Inconsistent structure	add logging
RB-TreeAga (2)	42	×	rbtree_map.c:174	C-A	Missing logging in a transaction	Inconsistent structure	add logging
	43	×	rbtree_map.c:355	C-A	Missing logging in a transaction	Inconsistent structure	add logging
Hashmap-TX (1)	44	✓	hashmap_tx.c:281	C-O	Use-after-free	Unexpected op failure	copy before free
Hashmap-atomic (2)	45	×	hashmap_atomic.c:129	C-A	Atomicity when creating hashmap	Inconsistent structure	logging/transaction
	46	✓	hashmap_atomic.c:198	C-A	Atomicity when assign pool id and offset	Inconsistent structure	inconsistency-recoverable design
Memcached(1)	47	×	items.c:538	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives

NOTE: **C-O**: persistence order correctness bug **C-A**: persistence atomicity correctness bug

Table 9.2: List of persistence correctness bugs discovered by Witcher. All 47 bugs have been confirmed by authors or existing tools, and 36 of 47 bugs are new. There are 25 persistence ordering bugs and 22 persistence atomicity bugs. One bug (ID 1) is in the PMDK library.

Case studies. Witcher detects many critical and sophisticated bugs. For instance, Bug ID 1 was a persistence ordering bug in PMDK's persistent pool allocator `pmemobj_tx_zalloc`, classified as "Priority 1: showstopper" [3]. The bug did not manifest in other TX-PMDK applications as it resides in a code path that requires a large-size object allocation. As another example, the bug in CLHT (Bug ID 30) only occurs when a program crashes at a specific moment during rehashing while leaving a specific set of stores unpersisted.

Fixing persistence ordering bugs. Witcher detected 25 persistence ordering bugs in

total. 14 persistence ordering bugs occurred because developers did not add persistence primitives (`flush`/`fence`) or passed incorrect addresses as parameters. Fixing these bugs is straightforward. The rest of the 11 persistence ordering bugs had persistence primitives, but they persisted multiple stores in an incorrect order. Fixing them requires reordering persistence primitives. For example, Bug ID 1 in PMDK's pool allocator and Bug ID 7 in `level_insert` (Figure 5.1(b)) were fixed by reordering source codes [3, 5].

Fixing persistence atomicity bugs. WITCHER detected 22 persistence atomicity bugs in total. Four cases (Bug IDs 40-43) were a missing logging problem in transactional programs. Fixing is relatively trivial – add logging. For the rest of the 18 bugs appearing in low-level NVM programs, all of them indeed required design or implementation-level changes. We observed the following four fixing strategies: (1) To merge multiple writes into *one word-size write* to guarantee atomicity [62]. (2) To make program *crash-inconsistency-tolerable* in which an operation that notices any inconsistent state fixes it on behalf of another operation. This is similar to the concurrent data structure's *helping mechanism* [23], where an operation started by one thread but failed is later completed by another thread. (3) To make program *crash-inconsistency-recoverable*. This solution introduces a recovery code that is executed after a crash and fixes any observed inconsistency. (4) To use *logging/transaction* techniques.

9.2.2 Statistics of Persistence Correctness Bug Detection

Table 9.3 also presents the detailed statistics of WITCHER. Across 20 NVM programs, when tested with 2,000 operations, WITCHER infers in total 639K (32K on average) likely-ordering conditions and 48K (2.4K) likely-atomicity conditions. WITCHER generated 1835K (92K) crash NVM images, 213K (11K) of which failed output equivalence checking.

WITCHER finally generated 765 bug reports clustered by operation type and execution path

	Name	Correctness		Total	Likely-Correctness Cond' Inference			Output Equivalence Checking			
		C-O	C-A		# ordering conditions	# atomicity conditions	execution time	# crash NVM images	# image w/ output mismatch	# cluster	execution time
Library	libpmemobj	1	0	**1**	-	-	-	-	-	-	-
NVM KV Index	WOART	0	1	**1**	7601	1126	31m29s	31859	26	8	7m53s
	WORT	0	0	**0**	15975	3423	26m28s	56265	1	1	9m14s
	Fast Fair	1	3	**4(2)**	413232	1201	22m6s	59644	46878	104	20m25s
	Level Hash	10	7	**17**	28080	1708	1h2m	55114	45263	33	1h32m
	CCEH	0	2	**2(1)**	8935	1839	28m48s	19141	860	5	59m29s
RECIPE	P-ART	0	2	**2**	4155	3570	3h53m	44243	41	8	11m37s
	P-BwTree	2	0	**2**	32945	5333	1h24m	38572	4826	80	1h26m
	P-CLHT	1	0	**1**	1580	364	1h23m	10370	476	3	27m27s
	P-CLHT-Aga	3	0	**3(1)**	8090	1084	55m49s	39918	248	5	1h43m
	P-CLHT-Aga-TX	2	0	**2(2)**	4358	477	2h	27949	242	5	49m25s
	P-Hot	3	0	**3**	20132	16403	5h10m	96295	905	155	21m35s
	P-Masstree	0	1	**1**	16139	2983	48m27s	115590	142	10	25m6s
PMDK	B-Tree	0	1	**1(1)**	1148	131	1h4m	114161	23255	46	20m15s
	C-Tree	0	0	**0**	9757	705	2h20m	30113	0	0	20m38s
	RB-Tree	0	1	**1(1)**	15342	726	1h24m	376891	5976	64	1h12s
	RB-Tree-Aga	0	2	**2(2)**	16188	725	1h23m	386252	82801	219	2h29m
	Hashmap-TX	1	0	**1**	8991	802	2h	30364	469	11	21m33s
	Hashmap-atomic	0	2	**2(1)**	7931	1078	2h	30068	272	8	1h22m
Server	Memcached	1	0	**1(1)**	11089	2746	1h12m	11348	0	0	1h29m
	Redis	0	0	**0**	7787	1270	6h49m	260526	0	0	3h3m
Total		**25(3)**	**22(8)**	**47(11)**	639455	47694	36h37m	1834683	212681	765	18h58m

C-O: persistence order correctness bug **C-A**: persistence atomicity correctness bug **(#)**: number of known bugs

Table 9.3: The tested NVM programs, the number of detected persistence correctness bugs, and the detailed statistics of Witcher bug detection.

(§5.2.4). To analyze the root cause of the correctness bugs and to communicate with the developers, we investigated all generated bug reports. Witcher provides sufficient information for root cause analysis, including execution trace, crash location, persisted and unpersisted writes, and a crash NVM image, which can be loaded for further `gdb` debugging. As the third-party tester, we could identify the root causes of detected correctness bugs from the Witcher's reports, manually but guided by `gdb`-based debugging. Multiple clusters shared the same root causes, and we reported and confirmed 25 persistence ordering bugs and 22 persistence atomicity bugs.

Table 9.3 reports testing time. Inferring likely-ordering/atomicity conditions took a few minutes to seven hours. Output equivalence checking took a few minutes to three hours, whose total cost is proportional to the number of tested crash NVM images and the cost of each test run. Testing `Memcached` and `Redis` based on live networking generally takes longer than the others. Note that Witcher systematically explores and validates feasible NVM

states (one by one) and thus it may take longer than other dynamic tools (*e.g.*, PMTest) testing one execution, yet it is much faster than other exhaustive testing tools (*e.g.*, Yat) thanks to pruning based on likely-ordering/atomicity conditions. We make the comparison in the following sections.

9.2.3 Scalability and Comparison with Yat

This section evaluates how effectively our likely-ordering/atomicity condition-based approach can prune the testing space, and thus improve scalability. First, we simulate the existing exhaustive-testing-based tool Yat [78] and compare the number of crash states that Yat will validate using the same trace with 2,000 random operations. Figure 9.1 shows the representative results for Level Hashing, FAST-FAIR, and CCEH programs. The test space of Yat is several orders larger than **Witcher**. Sudden spikes happen in Yat when there is a rehashing in Level Hashing and CCEH or a node split/merge in FAST-FAIR. **Witcher** only tests when there is a violation of likely-ordering/atomicity conditions, significantly reducing the number of test cases (yet detecting many bugs).

Second, Table 9.3 shows that with likely-ordering/atomicity conditions, **Witcher** tested 19K-60K NVM states for the three programs. Ideally, we wanted to test the entire NVM states and check if there is any bug that **Witcher** may miss. However, as shown in Yat simulation, the NVM state space is too huge to explore them all. Alternatively, we tested 100 million randomly chosen NVM states (without considering likely-ordering/atomicity conditions), which is 1677×-5224× larger NVM test space. Running 100M cases costs around one week for each program. The results show that the random 100M cases can only detect one or two of the bugs that **Witcher** detected, yet there was no new bug. Without a full search, we cannot conclude that likely-ordering/atomicity conditions are sound. However, the result shows that random pruning does not work, and our approach effectively detects many bugs.

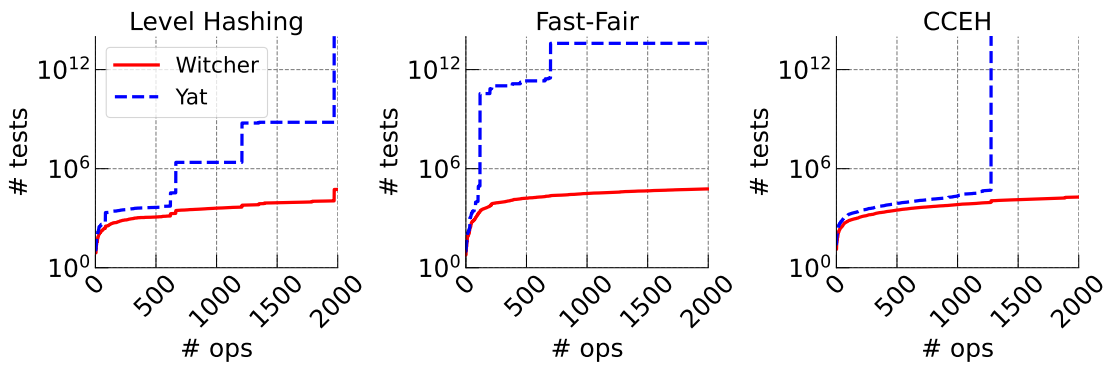


Figure 9.1: Test space comparison between Witcher and Yat for 2,000 random operations.

9.2.4 Bug Detection Effectiveness Comparison

We compared the persistence correctness bugs detected by Witcher, Agamotto, PMTest, and XFDetector. Making an apples-to-apples comparison among testing tools is hard with different test cases, testing resources and budgets, bug targets, etc. Therefore, we focus on checking if Witcher can detect the bugs that the others have found. In §9.2.1, we reported that Witcher discovered 36 new persistence correctness bugs.

Agamotto. We tested Agamotto with the same test cases (2,000 operations) used to evaluate Witcher. We set the memory resource as 32GB and the time limit as 24 hours for each Agamotto test. To detect PMDK transaction bugs, we enabled Agtamotto's custom checker. We evaluated B-Tree, RB-Tree, Hashmap-atomic, P-CLHT, Memcached and Redis including PMDK libraries. We used a modified version of Agamotto from the paper. To execute the same test cases, we asked the authors to support non-symbolic client connections for Memcached and Redis. We also asked them to fix a bug in the bug reporting logic. The modified Agamotto in our experiments found more bugs than the original paper.

Witcher detected all seven persistence correctness bugs detected by Agamotto. Agamotto missed two bugs (Bug IDs 1 and 46) due to the lack of program-specific oracles, showing the benefits of output equivalence checking.

PMTest and XFDetector. We also compared WITCHER with two annotation-based approaches. Seven programs were tested by WITCHER, PMTest, and XFDetector in common: B-Tree, C-Tree, RB-Tree, Hashmap-TX, Hashmap-atomic, Memcached and Redis. For correctness bugs, WITCHER detects three out of four bugs PMTest/XFDetector found in B-Tree (Bug ID 40), RB-Tree (Bug ID 41), and Hashmap-atomic (Bug ID 45). In addition, WITCHER detected three more new bugs (Bug IDs 1, 44 and 46), which were missed by PMTest/XFDetector.

WITCHER missed one bug in `Redis` reported by PMTest and XFDetector. The bug turns out to be benign. The bug is in the server initialization code. After allocating a PMDK root object, `Redis` initializes the root object to zero "outside" of a PMDK transaction. PMTest/XFDetector detects this unprotected update as a bug. However, this is benign – it does not lead to an inconsistent state. The root object was allocated using `POBJ_ROOT` [11], which already zeroed out the newly allocated object. Both the old and new values are zero. Therefore, it does not matter if the new zero update is persisted or not. WITCHER actually detected this store violating a likely-atomicity condition, and performed output equivalence checking. But it does not show any visible divergence. This example particularly shows the benefit of our output equivalence checking, pruning false positives.

Summary. WITCHER is able to detect all the known persistence correctness bugs and identify new persistence correctness bugs as well. WITCHER uses program-agnostic rules to find persistence correctness bugs from log-free NVM programs. WITCHER detects a new group of program-specific persistence correctness bugs, which cannot be detected by previous works because of the lack of program-specific oracles. WITCHER's efficiency could be further improved if integrated with a smart test case generator (*e.g.*, fuzzing, symbolic execution), with which new program paths can be explored, or the same program paths can be achieved with simpler test cases.

9.2.5 Testing Non-Key-value Store NVM Programs

We extended Witcher for testing a persistent array [7] and a persistent queue [9] from PMDK to demonstrate the feasibility of applying Witcher to non-key-value NVM programs. The persistent array supports allocation, reallocation, deallocation, and print operations. The persistent queue supports enqueue, dequeue, and print operations. We extended our template driver to support these non-key-value operations. For output equivalence checking, Witcher leverages outputs from print operations, which list all data in an array or a queue. We redirect the output of each operation to an output file to check if the test and oracle executions produce the same outputs. Similar to previous experiments, Witcher tested them using test cases with randomly generated 2,000 operations. Witcher detected one (known) correctness bug [8] in the persistent array.

9.3 Detecting Persistence Correctness Bugs by Inferring Likely-linearization Points

This section presents the following experiment results for Durinn.

- We report and analyze the persistence correctness bugs detected by Durinn (§9.3.1) along with detailed statistics, including the number of tests and testing time (§9.3.2).
- We evaluate the effectiveness and (empirical) soundness of Durinn's likely-linearization inference technique (§9.3.3).
- We compare Durinn with other NVM crash-consistency testing tools in terms of bug detection effectiveness and test space reduction (§9.3.4).

9.3.1 Detected Persistence Correctness Bugs

In summary, DURINN detected 27 (15 new) persistence correctness bugs from 12 concurrent 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 9.4 shows the source code locations, impacts and fix strategies of the detected bugs.

(DL1) Incompletely-Durable bugs. DURINN detected 10 Incompletely-Durable bugs. Figure 6.4(a) discussed in §6.1.1 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 6.4(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 6.4(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`

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 [13]
	16		✓	N4.cpp:17	DL2	Atomicity between metadata and key-value	Unable to recover	inconsistency-tolerable design [13]
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 [12]
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 [12]
	22		✓	HOTRowexNode.hpp:315	DL1	Missing persistence primitives	Points to garbage	add persistence primitives [12]
	23		✓	HOTRowex.hpp:270	DL1	Missing persistence primitives	Points to garbage	add persistence primitives [12]
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 9.4: List of persistence correctness bugs detected by Durinn. In total, 27 (15 new) persistence correctness bugs were detected from 12 concurrent NVM data structures. There were 10 Incompletely-Durable bugs, 7 Unrecovered-Durable bugs, and 10 Visible-But-Not-Durable bugs.

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.

9.3.2 Statistics of Persistence Correctness Bug Detection

Table 9.5 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

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 9.5: The tested NVM programs, the number of detected persistence correctness bugs, and the detailed statistics of DURINN bug detection.

likely linearization points. On average, using static analysis described in §6.2.2, 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 §9.3.3.

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 minutes) 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 9.4. 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.

9.3.3 Likely-Linearization Point Inference

To infer likely-linearization points, Durinn uses the two heuristics, *i.e.*, *Guarded-Protection* and *Publish-after-Initialization*, described in §6.2.2. 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 9.2 shows the effectiveness of the proposed likely-LP inference techniques, compared to the manually identified LPs. The first bar represent the total number of NVM stores in the trace. 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

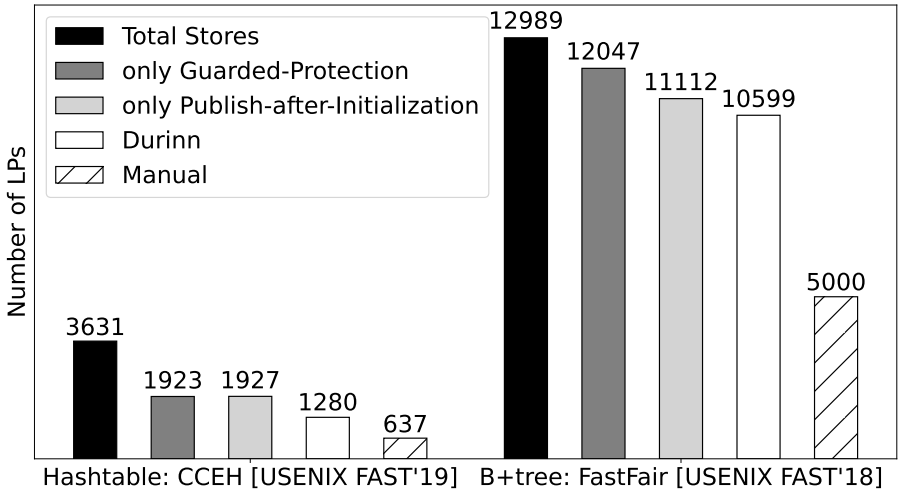


Figure 9.2: A case study of likely-linearization point inference.

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 9.5, CCEH and Fast Fair are the most difficult data structures in terms of the reduction ratio between the stores and the likely-LPs.

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.

9.3.4 Comparison with Other Tools

We present the detailed comparison with WITCHER [45], the state-of-the-art NVM crash-consistency bug detector, and Yat [78], an exhaustive crash-consistency testing tool.

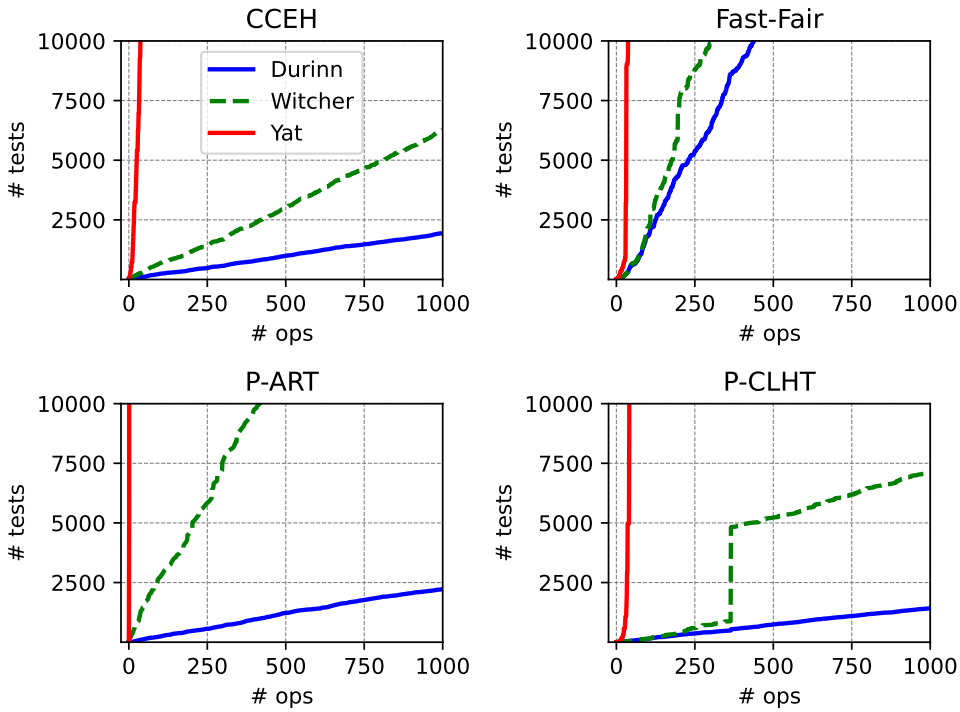


Figure 9.3: Test space comparison.

Bug Detection. We compared the bug detection effectiveness with **Witcher**. **Witcher** claims that it can detect all the persistence correctness 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 performed by **Durinn**, Yat, and **Witcher** over the same 1000 operations. Figure 9.3 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 two 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.

Discussion. WITCHER infers likely-ordering/atomicity conditions by analyzing program dependencies among NVM accesses. WITCHER issues a test whenever there is a likely-ordering/atomicity violation between two NVM accesses, That is, WITCHER tests every NVM accesses to explore all possible operation behaviors. Exploring all possible behaviors is expensive, especially when testing multi-threaded programs. On the other hand, DURINN infers likely-linearization points to understand operation behavior. By understanding operation behaviors, DURINN only tests potential incorrect behaviors, so that only certain NVM accesses need to be tested. Specifically DURINN only adversarially tests worst-case scenarios for the three DL bug patterns derived based on durable linearizability model. Thus, DURINN significantly reduces the test space of crash state further, and also reduces the test space of thread interleaving.

9.4 Detecting Persistence Performance Bugs by Analyzing Program Trace

We evaluated our trace-based persistence performance bug detection with the same programs and test cases used by WITCHER. Table 9.6 shows that our trace-based solution detected 158 performance bugs from PMDK library and tested programs in total and 113 of them are new bugs. We detected 102 unpersisted performance bugs; 21 extra flush performance bugs; 15 extra fence performance bugs; and 20 extra logging performance bugs, as classified in §3.1.2.

	Name	Performance				Total
		P-U	P-EFL	P-EFE	P-EL	
Library	libpmemobj	5	0	0	0	**5**
NVM KV Index	WOART	1	2	3	0	**6**
	WORT	1	1	0	0	**2**
	Fast Fair	5	0	1	0	**6**
	Level Hash	11	12	0	0	**23**
	CCEH	8	1	1	0	**10**
RECIPE	P-ART	9	0	1	0	**10**
	P-BwTree	1	0	1	0	**2**
	P-CLHT	7	0	1	0	**8**
	P-CLHT-Aga	10	0	1	0	**11**
	P-CLHT-Aga-TX	10	4	1	2	**17(15)**
	P-Hot	0	0	4	0	**4**
	P-Masstree	5	0	1	0	**6**
PMDK	B-Tree	0	0	0	5	**5(5)**
	C-Tree	0	0	0	0	**0**
	RB-Tree	0	0	0	0	**0**
	RB-Tree-Aga	0	0	0	13	**13(13)**
	Hashmap-TX	0	0	0	0	**0**
	Hashmap-atomic	0	0	0	0	**0**
Server	Memcached	29	1	0	0	**30(11)**
	Redis	0	0	0	0	**0**
Total		**102(17)**	**21(8)**	**15(2)**	**20(18)**	**158(45)**

P-U: unpersisted performance bug **P-EFL**: extra flush performance bug
P-EFE: extra fence performance bug **P-EL**: extra logging performance bug
(#): number of known bugs

Table 9.6: The statistics of detected persistence performance bugs by our trace-based solution.

Compared against Agamotto, Agamotto and our solution both discovered 61 bugs in common. Our solution detected 9 unique bugs and Agamotto found 43 bugs. Recall that the performance bug detection depends on tested program paths. The result implies that our solution and Agamotto explored different program paths, showing the pros and cons of (guided) symbolic execution by Agamotto and our trace-based approach. 43 Agamotto-unique performance bugs were found in PMDK libraries that Agamotto's symbolic execution did explore but our solution did not.

Chapter 10

Discussion

This chapter discusses the soundness and the completeness of our proposed solutions (§10.1), how to extend our proposed solutions to test general NVM programs (§10.2), test case generation for persistence bug detectors (§10.3), how to extend our proposed solutions to test NVM programs running on event-driven architecture (§10.4), and the impacts of persistent cache for persistence bug detection (§10.5).

10.1 Soundness and Completeness

Our proposed solutions are not sound (may have false negatives) for three reasons. First, our proposed solutions are trace-based dynamic detectors that take test cases as input. Our proposed solutions may miss persistence bugs that did not appear in a trace. Second, the proposed likely-correctness inference is based on heuristics and may miss true correctness conditions in theory. Missing a correctness condition means no testing, so our proposed solutions may miss persistence bugs. Last, our proposed solutions do not explore all possible NVM states and thread interleaving. In theory, some more complex combinations of persisted and unpersisted stores may be required to trigger a persistent bug. Furthermore, triggering a persistence bug may need more than two concurrent thread interleaving. However, our empirical study (§9.2.4 and §9.3.4) shows that both Witcher and Durinn detects all the persistence correctness bugs reported by existing bug detectors and indeed found more new

bugs without a test space explosion problem and user-provided oracles.

On the other hand, our proposed solutions are complete for a given trace under test (do not have false positives) as we perform durable linearizability validation. Any constructed crash NVM image that turns out to violate durable linearizability is indeed a definite clue of a true persistence correctness bug (by definition).

10.2 Testing General NVM Programs

The current prototypes are designed to test NVM programs in which (1) the granularity of "operation" and programming interfaces are well known (*e.g.*, insert, delete, etc.); and (2) durable linearizability is used as a correctness criterion. Testing general NVM programs requires a user to define its own operation granularity and create a deterministic test case for durable linearizability validation. For instance, NVM-based file systems may use POSIX file I/O interfaces. Besides durable linearizability, our proposed solutions can be extended for other correctness criteria: *e.g.*, buffered durable linearizability [64], and strict serializability for transactional programs [109]. These criteria produce different sets of oracles to compare during consistency validation.

10.3 Test Case Generation

The proposed techniques in this dissertation assume a test case, which is a sequence of operations to be executed by the testing program, is available. Generating test cases revealing NVM bugs could be a promising future work. PMFuzz [86] is the only available input generator for persistence bug detectors. PMFuzz employs fuzzing technique to generate inputs, each of which includes one normal or crash NVM image and a sequence of program

commands. However, PMFuzz only considers the coverage of program paths that contain NVM accesses as the fuzzing feedback. It would be an interesting research direction to leverage inferred implicit hints from NVM program as the fuzzing feedback.

10.4 Inferring Likely-correctness Conditions from NVM Programs Running on Event-driven Architecture

In this dissertation, our proposed solutions target only conventional thread-based NVM programs. We believe in the arrival of event-based NVM programs in the near future. Event-driven architecture (EDA) has been widely adopted in modern mobile systems, web servers and IoT alike, *e.g.*, Android [1] for mobile systems and Node.js [120] for web servers. We believe that detecting persistence bugs from event-driven NVM programs brings broad impacts in the future and is a promising research direction.

However, it is challenging to infer likely-correctness conditions from NVM programs running on EDA. In EDA, events are handled sequentially by event loop(s), and long-running tasks are offloaded to other threads. Due to the hybrid concurrency model in EDA, the non-determinism comes from not only non-deterministic thread schedule but also non-deterministic event posting order [20, 32, 51, 91]. Event-based and thread-based programming models have distinct patterns and dissimilar happens-before relations [77], making it hard to detect them together.

We argue that the *threadification* [44] technique enables to infer likely-correctness conditions from event-driven NVM programs. The key idea of threadification is to *model events as threads*. Threadification converts happen-before relations between events into happen-before relations events. More specifically, threadification models an event-driven NVM program as

a conventional thread-based NVM program. In this way, we can apply the proposed likely-correctness condition inference for thread-based NVM programs after the threadification.

10.5 Persistent Cache

The future generation of Intel architecture is expected to adopt eADR support (Extended Asynchronous DRAM Refresh) [60] that includes a cache into the persistent domain. For an eADR-enabled Intel architecture, there will be no gap between a linearization point and a durability point because once the effect of a store reaches a cache, it is guaranteed to be written back to the NVM. This means that as long as a data structure is linearizable, persistence ordering bugs would not appear. On the other hand, persistence atomicity bugs are still relevant as they require 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.

We believe that the architectural support for persistent cache such as eADR is the right direction in terms of improving the programmability and reliability of NVM programs. However, eADR is not readily available yet. It is also not clear if eADR-like persistent cache design can be and will be adopted by other architectures such as ARM, MIPS, RISC-V. In particular, for some energy-critical computing domains such as battery-less IoT systems, prior study [131] started questioning the energy efficiency of persistent cache. Furthermore, the next-generation Compute Express Link (CXL) [30] standard presents a cache-coherent interconnect for processors, memory expansion and accelerators. With CXL, it is expected that NVM will be attached to the PCIe bus as a memory device with a volatile cache (write buffer). To control durability, CXL will provide Global Persistent Flush (GPF) to coordinate a global flush activity between the host and the CXL domain.

The energy efficiency and scalability of eADR-like design require further study. We hope that the lessons studied in this dissertation shed light on the potential hazard in NVM programming in general, and guide the future NVM architectures, programming models, and tools.

Chapter 11

Related Work

Since we have discussed the existing persistence bug detectors in §3.3, this chapter introduces the related works in a broad scope. In this chapter, we first introduce persistent indexes (§11.1). We then introduce NVM optimized logging and FASE techniques for achieving crash-consistency (§11.2). We further introduce linearizability testing for concurrent programs (§11.3) and related works employing likely-correctness conditions (§11.4) to detect other types of bug or inconsistency. Lastly, we introduce crash consistency testing in file systems (§11.5) and other concurrency testing techniques (§11.6).

11.1 Persistent Indexes

PACTree [73] is a high-performance persistent range index following its proposed *Packed Asynchronous Concurrency* guidelines. The key idea of the guidelines is to access NVM is a packed manner and exploit asynchronous concurrency control. TIPS [110] is a framework to systematically make volatile indexes persistent. TIPS adopts a novel DRAM-NVM tiering to support index-agnostic conversion and durable linearizability.

There have been various research efforts to design efficient B+-tree-based persistent indexes [18, 28, 34, 55, 81, 102, 133] and trie-based persistent indexes [79, 80, 90]. BzTree [18] is a lock-free B+-tree. For lock-free implementation, it relies on Persistent Multi-word Compare-And-Swap (PMwCAS) [123] primitive, which supports atomic and crash-consistent

multi-word updates. FP-tree [102] is a DRAM-NVM hybrid B+-tree which places reconstructable internal nodes on faster DRAM. LB+-tree [81] optimizes the FP-tree design for OptaneNVM by leveraging an XPLine size granularity to avoid write amplification. ROART [90] is an improved version of P-ART [80] for supporting efficient range queries, lower memory allocation overhead, and correctness.

11.2 NVM Optimized Logging and FASE Techniques

There have been various research efforts to optimize the logging protocols for NVM [17, 53, 66, 70, 72, 99, 100, 105, 106, 115, 122]. To reduce the durability cost and hide the persistence latency, asynchronous commit policies [53, 99, 122] have been proposed. Some studies [17, 66, 106] leverage NVM to correctly restore the partial disk writes upon recovery. Timestone [76] is a highly scalable durable transactional memory system based on multi-version concurrency control. It proposes a multi-layered hybrid DRAM-NVM logging scheme to significantly reduce write amplification in NVM.

Another research direction of achieving crash-consistency for lock-based NVM programs is to leverage failure-atomic critical section (FASE) [24, 46, 52, 74, 83] to guarantee persistence atomicity at the granularity of a critical section. For example, JUSTDO [63] and iDO logging [83] enable a crashed NVM program to recover by resuming the execution of the incomplete FASE upon the crash.

11.3 Linearizability Testing

Line-up [21] is the first complete and automatic checker for deterministic linearizability. It detects thread-safety violations by comparing the concurrent execution to linearizable exe-

cutions of a test. Similarly, Round-up [132] 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.* [107] 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.

11.4 Likely-Correctness Conditions

Prior works have used a concept of likely-correctness conditions to detect program bugs [39, 69, 75, 88, 94, 130], to verify the network [87], and to identify resource leaks [121]. Engler *et al.*'s version (called beliefs) [39] enables automatic analysis of likely correctness conditions without in-depth knowledge.

11.5 Crash Consistency Testing in File Systems

There has been a long line of research in testing and guaranteeing crash consistency in file systems [25, 26, 48, 71, 95, 111, 117, 126, 127, 128]. In-situ model checking approaches such as EXPLODE [128] and FiSC [127] systematically test every legal action of a file system. B3 [95] performs exhaustive testing within a bounded space, which is heuristically decided based on the bug study of real file systems. Feedback-driven File system fuzzers, such as Janus [126] and Hydra [71], mutate both disk images and file operations to thoroughly explore file system states.

11.6 Other Concurrency Testing Techniques

AtomFuzzer [104] is a randomized active atomicity violation detector, which modifies the thread scheduler behavior to create atomicity violations with high probability. RaceFuzzer [114] 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 [41] uses adversarial memory to classify race conditions as destructive or benign on systems with relaxed memory models. Relaxer [22] detects sequential consistency violations in a relaxed memory model by actively leading execution to predicted violations.

An iterative context bound [96] or delay bound [38] has been used to (unsoundly yet effectively) prune the thread-interleaving test space when testing multi-threaded programs.

Chapter 12

Conclusion

Providing debugging support for software development is a critical research topic in modern computing systems. This dissertation specifically addresses the challenge of detecting persistence bugs within NVM programs. This dissertation proposes to infer likely-correctness conditions in order to detect persistence bugs from NVM programs in a scalable and automatic manner. Using inferred likely-correctness conditions is a general approach and is not limited to only detecting persistence bugs from NVM programs. We believe the proposed solutions of inferring likely-correctness conditions could be applied to detecting new types of bugs in future software with novel hardware, operating systems, or programming languages. We believe the proposed solutions in this dissertation can lay the foundations for future software debugging research and help our community to build bug-free software.

We briefly summarize our proposed solutions as follows.

We propose **Witcher**, which infers likely-ordering/atomicity conditions to detect persistence bugs from NVM programs. **Witcher** infers likely-ordering/atomicity conditions to effectively explore the NVM state test space of a program and perform output equivalence checking to identify an incorrect execution without user-provided test oracles. To the best of our knowledge, **Witcher** is the first persistence bug detector that uses program-agnostic rules to find persistence correctness bugs from log-free NVM programs. In our evaluation, **Witcher** detects 47 (36 new) correctness bugs in NVM-backed key-value stores and the PMDK library. **Witcher** does not suffer from test space explosion nor does it require manual test oracles to detect these bugs.

We propose DURINN, which infers likely-linearization points to detect persistence correctness bugs within NVM programs. To the best of our knowledge, DURINN is the first durable linearizability checker for concurrent NVM data structures. We first perform a detailed analysis of how a linearizable data structure may violate durable linearizability. From this analysis, we derive three durable linearizability bug patterns that render a linearizable data structure not durably linearizable. We then propose adversarial crash state and thread interleaving construction and likely-linearization point inference to allow DURINN to detect persistence correctness bugs in an active and scalable manner. In our evaluation, DURINN reports 27 (15 new) persistence correctness bugs and outperforms state-of-the-art NVM testing tools in terms of bug detection effectiveness and test space reduction.

We use a trace-based approach to detect persistence performance bugs. We leverage the collected dynamic program trace and detect persistence performance bugs during NVM persistence simulation. Our trace-based persistence performance bug detection detected 158 persistence performance bugs in total, of which 113 are new bugs.

Bibliography

[1] The Android mobile operating system. https://www.android.com/.

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

[3] PMDK issue to fix reported bug in allocation. https://github.com/pmem/pmdk/issues/4945.

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

[5] Level Hashing commit to fix reported bugs. https://github.com/Pfzuo/Level-Hashing/commit/5a6f9c111b55b9ae1621dc035d0d3b84a3999c71.

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

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

[8] Detected correctness bug in the persistent array, . https://github.com/pmem/pmdk/issues/4927.

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

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

[11] PMDK Root Object APIs. https://pmem.io/pmdk/manpages/linux/master/libpmemobj/pmemobj_root.3.

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

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

[14] 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.

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

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

[17] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind Logging. In *Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB)*, New Delhi, India, March 2016.

[18] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. In *Proceedings of the 44th International Conference on Very Large Data Bases (VLDB)*, Rio de Janeiro, Brazil, August 2018.

[19] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 677–694, Amsterdam, Netherlands, October 2016.

[20] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. In *Proceedings of the 26th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 332–348, Pittsburgh, PA, USA, 2015.

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

[22] 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.

[23] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 34th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 241–250, Donostia-San Sebastián, Spain, July 2015.

[24] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 25th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, October 2014.

[25] 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 ACM Symposium on Operating Systems Principles (SOSP), pages 18–37, Monterey, CA, October 2015.

[26] 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 ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017.

[27] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, Hawaii, USA, September 2015.

[28] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, pages 2634–2648, Virtual, August 2020.

[29] 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.

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

[31] 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.

[32] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 145–160, Belgrade, Serbia, April 2017.

[33] Anthony Demeri, Wook-Hee Kim, Madhava Krishnan Ramanathan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. Poseidon: Safe, Fast and Scalable Persistent Memory Allocator. In *Proceedings of the 21st International Middleware Conference (Middleware)*, pages 207–220, Virtual, December 2020.

[34] Solar Designer. Bugtraq: Getting around non-executable stack (and fix), 1997. https://seclists.org/bugtraq/1997/Aug/63.

[35] 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.

[36] 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)*, page 478–493, Huntsville, Ontario, Canada, 2019.

[37] 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.

[38] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, page 411–422, Austin, Texas, USA, 2011.

[39] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72, Chateau Lake Louise, Banff, Canada, October 2001.

[40] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. In *Proceedings of the ACM Transactions on Programming Languages and Systems*, 1987.

[41] 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.

[42] 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.

[43] 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)*, Virtual, June 2020.

[44] Xinwei Fu, Dongyoon Lee, and Changhee Jung. nadroid: statically detecting ordering violations in android applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, pages 62–74, Vienna, Austria, February 2018.

[45] 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.

[46] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for Synchronization-free Regions. In *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, June 2018.

[47] 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.

[48] 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, San Jose, CA, February 2008.

[49] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. Efficient construction of program dependence graphs. *ACM SIGSOFT Software Engineering Notes*, 18(3):160–170, 1993.

[50] 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.

[51] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 326–336, Edinburgh, United Kingdom, 2014.

[52] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[53] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, pages 389–400, Hangzhou, China, September 2014.

[54] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[55] 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.

[56] Intel. C++ bindings for libpmemobj (part 6) - transactions, 2016. URL http://pmem.io/2016/05/25/cpp-07.html.

[57] INTEL. Persistent Memory Development Kit, 2019. http://pmem.io/.

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

[59] INTEL. PMDK man page: pmemobj_open, 2020. https://pmem.io/pmdk/manpages/linux/master/libpmemobj/pmemobj_open.3.

[60] 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.

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

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

[63] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[64] 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.

[65] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 783–798, Renton, WA, July 2019.

[66] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. Scalable database logging for multicores. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, pages 135–148, TU Munich, Germany, August 2017.

[67] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 494–508, Ontario, Canada, October 2019.

[68] Sudarsun Kannan, Moinuddin Qureshi, Ada Gavrilovska, and Karsten Schwan. Energy Aware Persistence: Reducing Energy Overheads of Memory-based Persistence in NVMs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 165–177, New York, NY, USA, November 2016. ACM.

[69] Samantha Syeda Khairunnesa, Hoan Anh Nguyen, Tien N. Nguyen, and Hridesh Rajan. Exploiting implicit beliefs to resolve sparse usage problem in usage-based specification mining. In *Proceedings of the 28th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver Canada, October 2017.

[70] Junghoon Kim, Changwoo Min, and Young Ik Eom. Reducing excessive journaling overhead with small-sized NVRAM for mobile devices. *IEEE Transactions on Consumer Electronics*, 60(2):217–224, 2014.

[71] 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 (SOSP)*, pages 147–161, Ontario, Canada, October 2019.

[72] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[73] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, October 2021.

[74] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level Persistency. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 2018.

[75] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 161–176, Seattle, WA, November 2006.

[76] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, April 2020.

[77] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782.

[78] 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.

[79] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February–March 2017.

[80] 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.

[81] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Virtual, August 2020.

[82] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[83] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, Fukuoka, Japan, October 2018.

[84] 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.

[85] 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.

[86] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. Pmfuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

[87] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 499–512, Oakland, CA, May 2015.

[88] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of 21st ACM Symposium on Operating Systems Principles (SOSP)*, page 103–116, Stevenson, WA, 2007.

[89] M. Seltzer and V. Marathe and S. Byan. An NVM Carol: Visions of NVM Past,

Present, and Future. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 15–23, Paris, France, April 2018.

[90] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. Roart: Range-query optimized persistent art. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, Virtual, February 2021.

[91] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 316–325, Edinburgh, UK, June 2014.

[92] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *Proceedings of the 17th Workshop on Hot Topics in Storage and File Systems*, Santa Clara, CA, July 2017.

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

[94] 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 ACM Symposium on Operating Systems Principles (SOSP)*, pages 361–377, Monterey, CA, October 2015.

[95] 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.

[96] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 446–455, San Diego, CA, USA, 2007.

[97] 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.

[98] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1047–1064, November 2020.

[99] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *Proceedings of the 24th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, New Orleans, Louisiana, February 2018.

[100] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. SQLite Optimization with Phase Change Memory for Mobile Applications. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, pages 1454–1465, Hawaii, USA, September 2015.

[101] Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. *ACM Sigplan Notices*, 19(5):177–184, 1984.

[102] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Stor-

age Class Memory. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA, June 2016.

[103] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory Management Techniques for Large-scale Persistent-main-memory Systems. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, TU Munich, Germany, August 2017.

[104] 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.

[105] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. SQL Statement Logging for Making SQLite Truly Lite. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, pages 513–525, TU Munich, Germany, August 2017.

[106] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage management in the nvram era. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, pages 121–132, Riva del Garda, Italy, August 2013.

[107] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 521–530, Beijing, China, June 2012.

[108] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Austin, TX, USA, June 2009.

[109] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of armv8 and transactional models. In *Proceedings of the 30th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Athens, Greece, October 2019.

[110] Madhava Krishnan Ramanathan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. TIPS: making volatile index structures persistent with DRAM-NVMM tiering. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 773–787, July 2021.

[111] 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.

[112] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 139–152, Houston, TX, March 2013.

[113] David Schwalb, Tim Berning, Martin Faust†, Markus Dreseler, and Hasso Plattner†. nvm malloc: Memory Allocation for NVRAM. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, Hawaii, USA, September 2015.

[114] 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.

[115] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[116] 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.

[117] 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.

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

[119] Tom Talpey and Andy Ruddof. Advanced Persistent Memory Programming: Local, Remote and Cross-Platform. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018. URL https://www.usenix.org/conference/fast18/training-program.

[120] Stefan Tilkov and Steve Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[121] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32nd IEEE International Conference on Software Engineering (ICSE)*, pages 535–544, Cape Town, South Africa, May 2010.

[122] Tianzheng Wang and Ryan Johnson. Scalable Logging Through Emerging Non-volatile

Memory. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, Hangzhou, China, September 2014.

[123] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, Paris, France, April 2018.

[124] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[125] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2016.

[126] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, pages 818–834, San Francisco, CA, May 2019.

[127] 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.

[128] 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.

[129] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. Deuce: Write-efficient encryption for non-volatile memories. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[130] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, pages 363–378, Austin, TX, August 2016.

[131] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. Replaycache: Enabling volatile cachesfor energy harvesting systems. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 170–182, 2021.

[132] 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)*, page 4–14, Palo Alto, CA, 2013.

[133] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. Dptree: differential indexing for persistent memory. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, pages 421–434, Los Angeles, CA, August 2019.

[134] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, October 2018.