WITCHER: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores

Xinwei Fu
Virginia Tech

Wook-Hee Kim
Virginia Tech

Ajay Paddayuru Shreepathi
Stony Brook University

Mohannad Ismail
Virginia Tech

Sunny Wadkar
Virginia Tech

Dongyoon Lee
Stony Brook University

Changwoo Min
Virginia Tech

Abstract

The advent of non-volatile main memory (NVM) enables the development of crash-consistent software without paying storage stack overhead. However, building a correct crash-consistent program remains very challenging in the presence of a volatile cache. This paper presents WITCHER, a systematic crash consistency testing framework, which detects both correctness and performance bugs in NVM-based persistent key-value stores and underlying NVM libraries, without test space explosion and without manual annotations or crash consistency checkers. To detect correctness bugs, WITCHER automatically infers *likely correctness conditions* by analyzing data and control dependencies between NVM accesses. Then WITCHER validates if any violation of them is a true crash consistency bug by checking *output equivalence* between executions with and without a crash. Moreover, WITCHER detects performance bugs by analyzing the execution traces. Evaluation with 20 NVM key-value stores based on Intel's PMDK library shows that WITCHER discovers 47 (36 new) correctness consistency bugs and 158 (113 new) performance bugs in both applications and PMDK.

CCS Concepts: • **Hardware → Emerging technologies**; • **Software and its engineering → Software testing and debugging**.

Keywords: Non-volatile Memory, Crash Consistency, Debugging, Testing

ACM Reference Format:
Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. WITCHER: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores . In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483556

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483556

1 Introduction

Non-volatile main memory (NVM) technologies, such as Intel's Optane DC Persistent Memory [15, 61], is on the rise: *e.g.*, Google Cloud [2] and Aurora supercomputer [1]. NVMs provide persistence of storage along with traditional DRAM characteristics such as byte addressability and low access latency. The ability to directly access NVMs using regular `load` and `store` instructions provides a new opportunity to build *crash-consistent* software (*e.g.*, NVM-backed key-value stores) without paying storage stack overhead. Programs can recover a consistent state from a persistent NVM state in the event of a software crash, or a sudden power loss.

However, it is hard to design and implement a correct and efficient crash-consistent NVM program. NVM data on a volatile cache may not be persisted 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. The existing ISAs also do not support updating multiple NVM locations atomically.

To ensure crash consistency, the current NVM programming model requires (either application or library) developers to explicitly add a cache line flush and store fence instructions (*e.g.*, `clwb` and `sfence` in x86 architecture) and to devise a custom mechanism to enforce proper persistence ordering and atomicity guarantees. NVM programming thus becomes error-prone and misuse of NVM primitives may lead to correctness bugs (*e.g.*, misplaced flush/fence) or performance bugs (*e.g.*, redundant flush/fence). A correctness bug[1] is particularly critical as a program may lead to an inconsistent NVM state on a crash and fail to recover with permanent data corruption, irrecoverable data loss, etc.

Recently, several solutions have been proposed to detect persistence bugs in NVM programs. However, there are two critical issues, namely (1) scalability against testing possible

[1]In this paper, we refer to a crash consistency bug as a correctness bug to differ it from a performance bug, though it is one kind of correctness bugs.

NVM states and (2) the need to create manual test oracles to validate testing correctness. These issues still make it challenging to effectively test NVM-backed persistent key-value stores, one of the most prominent application domains among NVM software: *e.g.*, Level Hashing [88], RECIPE [54], Redis [13], and Memcached [10].

A line of testing tools that attempt to exhaustively test all possible NVM states (*e.g.*, Yat [52], PMReorder [39], and SCMTest [69]) has the potential to detect many bugs, but often suffer from test space explosion. To bound search cost, persistent key-value stores are typically backed by data structures with rebalancing operations (*e.g.*, rehashing in a hash table, split-merge in a B-tree). Triggering such operations and detecting persistence bugs therein require a test case with a large number of operations (a long execution), making exhaustive testing infeasible in practice.

Moreover, for a test oracle, existing NVM testing tools require users to provide a manually-designed, application-specific consistency checker to validate a program under test: *e.g.*, manual annotations in PMTest [57] and XFDetector [56], consistency checkers in Yat [52], application-specific oracles in Agamotto [67], ordering configurations in PMDebugger [26]. Persistent key-value stores often employ different forms of inconsistency tolerable and/or recoverable design. Devising application-specific test oracles not only requires significant manual efforts, but also is error-prone.

This paper presents Witcher, a new crash consistency testing framework that systematically explores the NVM state test space (without test space explosion) and automatically validates if each feasible NVM state is consistent or not (without manual test oracles). Witcher detects *application-specific correctness bugs* (*e.g.*, persistence ordering and atomicity violations) using *application-agnostic rules*. It can also identify performance bugs (*e.g.*, extra flush/fence) by analyzing the execution trace.

To address the test space challenge, Witcher infers a set of *likely-correctness conditions* that are believed true to be crash-consistent by analyzing program data/control dependencies among NVM accesses. Witcher then tests only those NVM states that violate the likely-correctness conditions, significantly reducing the NVM state test space. Though Witcher may unsoundly prune some NVM states to test, our evaluation shows that it can effectively prune the test space and detect many new correctness bugs. Since the *likely-correctness conditions* were derived from common NVM programming patterns, they are applicable to non-key-value applications. We show that Witcher detects crash consistency bugs in PMDK's persistent pool management codes (§7.2) and non-key-value applications (§7.7).

To mitigate the test oracle problem, Witcher employs *output equivalence checking* between program executions with and without a (simulated) crash. Persistent key-value stores 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 [44]. If a key-value store resuming from an NVM state that violates a likely-correctness 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. Our output equivalence checking requires determinism and checks for durable linearizability. Thus Witcher may not be applicable to NVM programs that permit non-deterministic output (*e.g.*, timestamp) or non-durable-linearizable behaviors.

We evaluated Witcher with 20 NVM-backed persistent key-value stores that are ported with Intel's PMDK library. Using randomly generated test cases with 2000 operations, Witcher detected 47 (36 new) correctness bugs and 158 (113 new) performance bugs in 20 programs. Witcher can detect bugs not only in applications but also in low-level PMDK libraries whose bug manifests as different outputs at the application level. For instance, Witcher finds a correctness bug in PMDK's persistent pool/heap management library, classified as "Priority 1: showstopper" [8]. All new correctness bugs are confirmed by the developers.

This paper makes the following contributions:

- We propose a new NVM software testing approach that infers likely-correctness 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 application-agnostic rules to find application-specific correctness bugs.
- Witcher detects 205 (149 new) correctness/performance 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.

2 Background

This section demonstrates the types of correctness bugs (§2.1) and performance bugs (§2.2) that Witcher found in crash-consistent NVM programs, along with real-world examples.

2.1 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.

(1) Persistence ordering violations. We found that many crash consistency and recovery mechanisms rely on a certain (application-specific) *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 [88] 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 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 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.

(2) Persistence atomicity violations. 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 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.

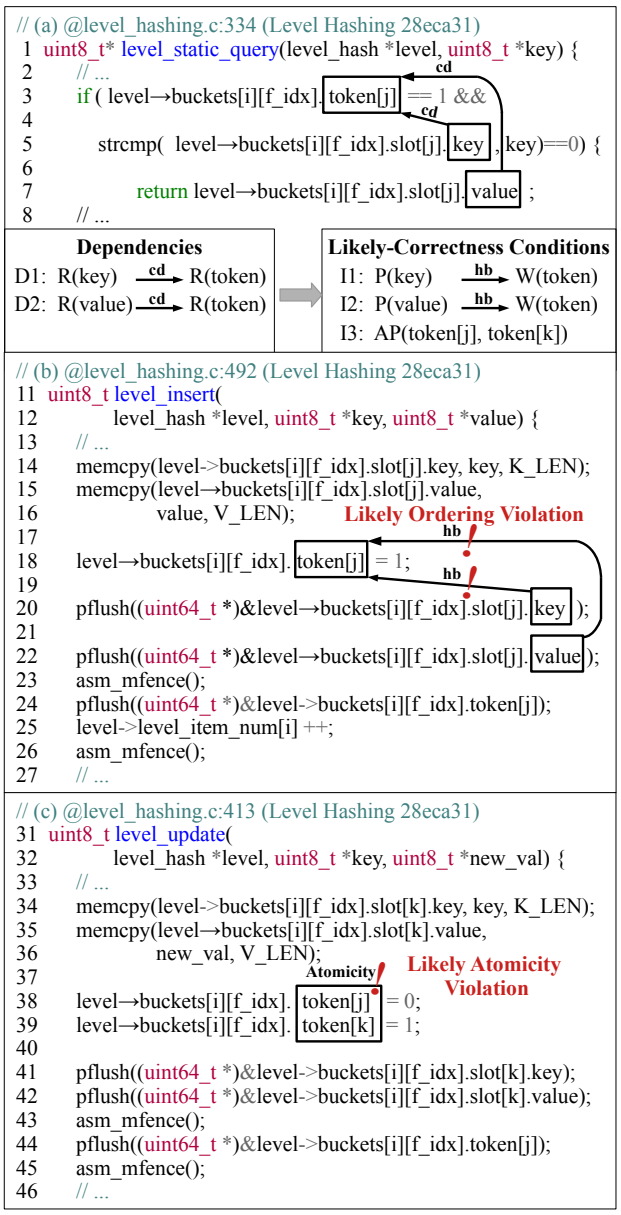

Figure 1. Using the likely-correctness conditions inferred from (a), Witcher finds two correctness bugs (b) and (c) in Level Hashing.

2.2 Performance Bugs

Previous studies [57, 67] found that performance bugs are prevalent in real-world NVM programs. Performance bugs do not cause an inconsistent state, yet it requires significant developer's time and effort to spot and fix them. Similar to prior work, we classify NVM 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 incurs higher latency than DRAM. 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. The extra can be removed without breaking 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 Overview of Our Approach

We first introduce how Witcher finds correctness bugs (§3.1) and performance bugs (§3.2). Then we will discuss how Witcher advances exiting approaches (§3.3).

3.1 Correctness Bug Finding

To detect correctness bugs, Witcher infers likely-correctness conditions (§3.1.1) and performs output equivalence checking to validate the NVM states violating them (§3.1.2).

3.1.1 Inference of Likely-Correctness Conditions. We propose a novel approach that analyzes program data/control dependencies among NVM accesses to infer likely-correctness 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 persistence ordering/atomicity conditions.

Using the aforementioned Level Hashing example, let us demonstrate how we can infer a likely-correctness condition from the query function `level_static_query` in Figure 1(a), and apply it to find the correctness bugs in `level_insert` and `level_update` in Figure 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 [4, 60, 82], logging implementations [16, 35, 36, 43, 51, 74, 81], persistent data structures [22, 24, 37, 49, 53, 54, 65, 71], memory allocators [17, 25, 38, 70, 77], and file systems [23, 27, 28, 46, 83]. 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 1(b)); (2) The reader checks if the flag is set before reading the key and value, which we call *"guarded read"* (Figure 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 1(a), we infer the first *likely persistence 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 persistence atomicity condition* – the updates of two or more guardians should be atomic (*i.e.*, `AP(token[j],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-correctness 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-correctness conditions to reduce NVM state testing space without manual annotations. In §4.2, we present more generalized rules to infer likely-correctness conditions beyond guarded protection. 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.

3.1.2 Validation with Output Equivalence Checking Witcher uses *output equivalence checking* to validate if an NVM state that violates an inferred likely-correctness condition is indeed inconsistent, indicating a crash consistency bug. Many NVM programs, including a persistent key-value store, aim to provide durable linearizability [44] 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-correctness 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-correctness condition is a true bug.

Output equivalence checking allows Witcher to automatically detect 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.

3.2 Dynamic Trace Based Performance Bug Finding

Witcher uses a *trace-based* approach to detect performance bugs. Unlike finding correctness bugs, which requires searching possible crashed NVM states, detecting performance bugs does not need crash simulation and only requires tracking the NVM persistence state in program order. For example,

	Test space exploration		Crash consistency validation (oracle)
	Input	NVM State	
Yat [52] PMReorder [39]	user-provided test case	exhaustive	user-provided oracle
Jaaru [32]	user-provided test case	model checking with pruning	visible manifestation
PMTest [57] XFDetector [56]	user-provided test case	manual annotation	user-provided oracle
Agamotto [67]	symbolic execution	PM-aware search algorithm	user-provided oracle
PMDebugger [26]	user-provided test case	user-provided oracle	user-provided oracle
Witcher (this work)	user-provided test case	systematic pruning based on likely-correctness conditions	output equivalence checking

Table 1. Comparison with existing crash consistency testing tools.

to detect an extra flush performance bug, the persistence state of the cacheline to be flushed before executing the flush instruction is needed. Witcher leverages the collected dynamic program trace and detects performance bugs during NVM persistence simulation.

3.3 Comparison with Existing Solutions

Table 1 summarizes how Witcher is different from existing crash consistency testing tools when it comes to detecting correctness bugs. For performance bugs, Witcher is similar to existing work, and §7.6 later shows the pros and cons of Witcher's trace-based approach, compared to the symbolic execution-based approach in Agamotto [67].

Exhaustive testing tools, such as Yat [52] and PMReorder [39], 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 permutations (see §7.5). Moreover, for each crashed state, they rely on a user-provided consistency checker to validate whether NVM data is consistent. However, the correctness of a manual checker is often a concern [45]. Recently, Jaaru [32], 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 still 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. Later in §7.5, we show that Witcher effectively prunes NVM state test space based on likely-correctness conditions.

The test space explosion problem motivated the *annotation-based approach*, such as PMTest [57] and XFDetector [56]. 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 violations such as Figure 1(c). XFDetector relies on user's manual investigation for validation. Agamotto [67] 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 app-specific correctness bugs (*e.g.*, persistence ordering/atomicity violations), Agamotto still requires users to provide test oracles. Similarly, PMDebugger [26] requires user-provided oracles (*i.e.*, ordering debugger configuration file) to detect app-specific correctness bugs. Later in §7.6, we show Witcher with output equivalence checking can detect the correctness bugs that the prior testing tools found and more without user-provided oracles or annotations.

4 Design of Witcher

Figure 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 (§4.1). For correctness bugs, Witcher infers likely-correctness conditions from the trace (§4.2), constructs a set of crash NVM images violating the likely-correctness conditions (§4.3), and performs output equivalence checking to validate if a likely-correctness condition violation is a true correctness bug (§4.4). Witcher analyzes the same trace to detect performance bugs as well (§4.5).

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, which will be further discussed in §5.2. This section assumes testing single-threaded programs.

4.1 Tracing Memory Accesses

Witcher instruments an NVM program using an LLVM compiler pass [14] and executes the instrumented binary with a test case to collect the execution trace. We trace load, store/non-temporal store[2] (including the updated value), branch, call/return, flush and memory fence instructions.

Suppose we trace Level Hashing in Figure 1 using the test case with four operations in Figure 3(a). Figure 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).

4.2 Inferring Likely-Correctness Conditions

Witcher correlates program data/control dependencies with NVM crash consistency correctness conditions. We first describe a set of inference rules for (1) likely persistence ordering conditions and (2) likely persistence atomicity conditions

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

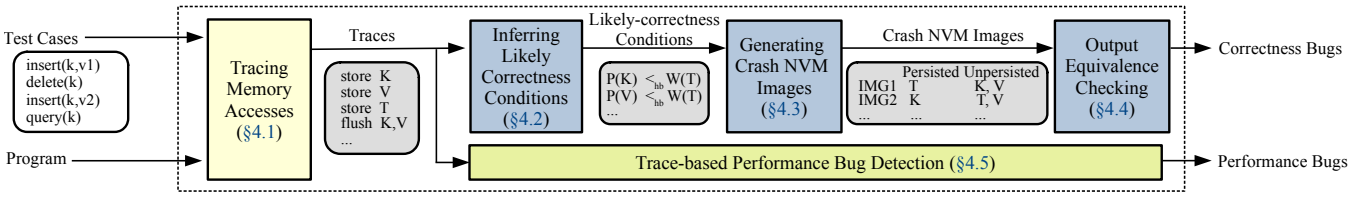

Figure 2. The architecture of Witcher. Witcher automatically detects (application-specific) correctness bugs (in blue) and performance bugs (in green) based on a given test case and its trace (in gray) without either manual annotation/oracle or exhaustive testing.

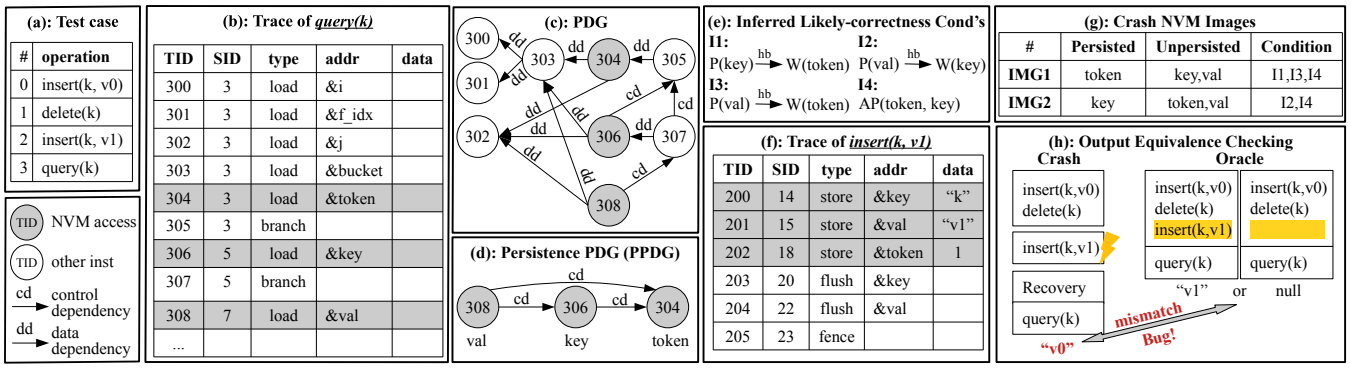

Figure 3. An example of Witcher's correctness bug detection steps.

(§4.2.1). Then we explain how Witcher uses program dependence analysis to infer the likely-correctness conditions from the trace (§4.2.2).

4.2.1 Inference Rules Table 2 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-correctness condition – *e.g.*, "`Y` is persisted, but `X` is not" (§4.3) and tests if the likely ordering/atomicity violation is a true crash consistency bug using output equivalence checking (§4.4). 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.

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

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

Table 2. The inference rules PO1–PO3 are for persistence ordering likely-correctness conditions and PA1 is for persistence atomicity.

(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 $W(Y) \xrightarrow{dd} R(X)$). 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.*, $P(X) \xrightarrow{hb} W(Y)$ where $\xrightarrow{hb}$ 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 2 says: for two NVM locations `X` and `Y`, if we find a Hint $W(Y) \xrightarrow{dd} R(X)$, we infer a likely PO condition $P(X) \xrightarrow{hb} W(Y)$. We later test an NVM state that violates the PO condition in which `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 $W(Y) \xrightarrow{cd} R(X)$, we infer a likely PO condition $P(X) \xrightarrow{hb} W(Y)$. 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 §3.1.1, 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 $R(Y) \xrightarrow{cd} R(X)$, we infer a Likely PO Condition $P(Y) \xrightarrow{hb} W(X)$. We note that here `X` is a guardian in

the guarded read pattern (*e.g.*, `token` in Figure 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 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.

4.2.2 Data/Control Dependence Analysis

Witcher performs program dependence analysis to infer likely-correctness conditions from the source codes and execution traces. Witcher first constructs a Program Dependence Graph (PDG) [31, 34, 68] 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-correctness condition inference rules. For example, Figure 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 (§4.1), 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 2. 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-correctness Condition. For example, the PPDG in Figure 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)`.

4.3 Generating Crash NVM Images

The next step is to generate a set of crash NVM images[3] that violate the likely-correctness conditions. Later in §4.4, we will describe how Witcher loads these NVM images and uses output equivalence checking to validate if a likely-correctness condition violation is a true bug or not.

At a high level, Witcher generates crash NVM images as follows. Witcher takes as input the same trace used to collect likely-correctness conditions and performs cache and NVM simulations along the trace. During the simulation, Witcher cross-checks if there is any violation of likely-correctness conditions. Each violating NVM state forms a crash NVM image to test. Witcher produces a set of crash NVM images for further validation.

4.3.1 Simulating Cache and NVM States

The goal of the cache/NVM simulations is to generate only feasible NVM states that violate likely-correctness 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 [52]. 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 [42, 78]).

Consider the trace of Level Hashing's `level_insert` code in Figure 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.

[3] In PMDK, an NVM image is a regular file containing an NVM heap state created, loaded, and closed by PMDK APIs [40].

4.3.2 Detecting Likely-Correctness Condition Violations

During the simulation, Witcher checks if there could be an NVM state that violates a likely-correctness 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 3(f) again. Before we execute the last `fence` instruction (TID 205), we check the four likely-correctness 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 that `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.

4.4 Output Equivalence Checking

Witcher validates the crash NVM images violating likely-correctness conditions and detects crash consistency bugs using output equivalence checking. In particular, Witcher focuses on testing durable linearizability [44]. 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 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.

4.5 Performance Bug Detection

Witcher detects the following performance bugs based on trace-based cache/NVM simulation. Witcher reports an unpersisted performance bug if a store still remains in the cache (not persisted) at the end of simulation yet it passes an output equivalence checking. When simulating a flush instruction, Witcher reports an extra flush performance bug if all prior stores have already been flushed by prior flush instructions. When simulating a fence instruction, Witcher reports an extra fence performance bug if there are no preceding flush instructions. For transactional NVM programs, Witcher reports an extra logging performance bug if a memory region or its subset has already been logged by preceding logging operation in the same transaction.

5 Discussion

5.1 Testing Non-Key-value Store NVM Programs

The current Witcher prototype is designed to test NVM-backed key-value stores 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 non-key-value NVM programs requires a user to define its own operation granularity and

create a deterministic test case for output equivalence checking. For instance, NVM-based file systems may use POSIX file I/O interfaces. Besides durable linearizability, Witcher can be extended for other correctness criteria: *e.g.*, buffered durable linearizability [44], and strict serializability for transactional programs [73]. These criteria produce different sets of oracles to compare during output equivalence checking.

5.2 Testing Multi-threaded NVM Programs

Witcher supports a limited form of testing for multi-threaded NVM programs. When testing multi-threaded programs, likely-correctness conditions can still be inferred with no modification. However, output equivalence checking requires two special considerations. First, the test case used for output equivalence checking should remain deterministic. This implies that the "prefix" test case before a crash is simulated during concurrent executions should be sequential (and produce deterministic outputs). Second, output equivalence checking should consider more oracles for multi-threaded cases. Each per-thread operation has two legal states (all or nothing), and we also need to consider different permutations of a linearization order. This implies that the cost of testing increases super-linearly. Witcher focuses on pruning the NVM state space in a systematic manner, and we leave thread-interleaving space reduction as future work.

6 Implementation

We built tracing and program dependency analysis based on Giri [76], a dynamic program slicing tool implemented in LLVM [14]. Our Giri modification comprises around 3,600 lines of C++ code. Other Witcher components are written in 4,400 lines of Python code. The Witcher prototype is available at https://github.com/cosmoss-vt/witcher.

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 [41]. Witcher runs PPDG construction, crashed NVM image generation, and output equivalence checking in parallel.

To support output equivalence checking, Witcher provides 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 Witcher checks if the test and oracle executions produce the same outputs.

7 Evaluation

7.1 Evaluation Methodology

Tested NVM programs. We evaluate Witcher with four groups of 20 (in total) real-world NVM-backed key-value stores (persistent indexes) (Table 3). The first group includes five highly optimized persistent key-value indexes, which are the backbone of many key-value stores and storage systems.

	Application	Version	Lib	Design	Core NVM Construct	Concurrency
NVM KV Index	WOART [53]	5b4cf3e	PMDK v1.8	LL	radix tree	ST
	WORT [53]	5b4cf3e	PMDK v1.8	LL	radix tree	ST
	Fast Fair [37]	c86f5fb	PMDK v1.8	LL	B+ tree	LB
	Level Hash [88]	28eca31	PMDK v1.8	LL	hash table	ST
	CCEH [65]	d53b336	PMDK v1.8	LL	hash table	LB
RECIPE	P-ART [54]	5b4cf3e	PMDK v1.8	LL	radix tree	LB
	P-BwTree [54]	5b4cf3e	PMDK v1.8	LL	B+ tree	LF
	P-CLHT [54]	5b4cf3e	PMDK v1.8	LL	hash table	LB
	P-CLHT-Aga [54]	53923cf	PMDK v1.8	LL	hash table	LB
	P-CLHT-Aga-TX [54]	53923cf	PMDK v1.8	TX	hash table	LB
	P-Hot [54]	5b4cf3e	PMDK v1.8	LL	trie	LB
	P-Masstree [54]	5b4cf3e	PMDK v1.8	LL	B tree + trie	LB
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
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 3. The description of tested NVM programs.

For high performance, they all have their own crash consistency mechanism using low-level (LL) persistence primitives such as `flush` and `fence` instructions. For example, FAST-FAIR [37] incorporates inconsistency tolerable design where a naive crash consistency bug detection approach would lead to false positives. The second group includes seven concurrent persistent indexes converted by RECIPE [54]. We used three different versions/configurations of P-CLHT to compare with Agamotto [67]. 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 six (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. We also note that `Memcached` and `Redis` maintain only a part of its application state in NVM as a persistent hash table, which turns out to be much simpler in design, compared to the other tested KV indexes.

All tested applications use PMDK library (`libpmemobj`) for persistent memory allocation or transaction. For some applications 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 traces and analyzes both applications and PMDK libraries such as persistence heap allocation and transactional undo logging logics.

Test cases. Witcher requires a deterministic test case such that it produces the same output for a given input for output equivalence checking (§3.1.2). Any deterministic test case with good code coverage would suffice. We leave a smarter test case generation (*e.g.*, fuzzing) as future work, and instead used random test case generation for well-known key-value

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 [8]
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 [11]
	27	✓	N4.cpp:17	C-A	Atomicity between metadata and key-value	Inconsistent structure	inconsistency-tolerable design [11]
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 assigin 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 4. List of 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.

interfaces such as `insert`, `delete`, `update`, `query`, and `scan`. Witcher randomly generates a list of operations, keys, and values. 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 2,000 randomly generated operations. We found that 2,000 operations are large enough to achieve a reasonable and stable code coverage (50%-80%) for our tested NVM programs. Missing code coverages are due to unused features (*e.g.*, garbage collection) and debugging codes.

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.

7.2 Detected Correctness Bugs

Witcher detected 47 (36 new) crash consistency bugs from 18 programs. There were 25 persistence ordering bugs and 22 persistence atomicity bugs. All the bugs were confirmed by the developers. Table 4 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 4) 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	Correctness		Performence				Total	Likely-Correctness Cond' Inference			Output Equivalence Checking			
		C-O	C-A	P-U	P-EFL	P-EFE	P-EL		# ordering conditions	# atomicity conditions	execution time	# crash NVM images	# image w/ output mismatch	# cluster	execution time
Library	libpmemobj	1	0	5	0	0	0	**6**	-	-	-	-	-	-	-
NVM KV Index	WOART	0	1	1	2	3	0	**7**	7601	1126	31m29s	31859	26	8	7m53s
	WORT	0	0	1	1	0	0	**2**	15975	3423	26m28s	56265	1	1	9m14s
	Fast Fair	1	3	5	0	1	0	**10(2)**	413232	1201	22m6s	59644	46878	104	20m25s
	Level Hash	10	7	11	12	0	0	**40**	28080	1708	1h2m	55114	45263	33	1h32m
	CCEH	0	2	8	1	1	0	**12(1)**	8935	1839	28m48s	19141	860	5	59m29s
RECIPE	P-ART	0	2	9	0	1	0	**12**	4155	3570	3h53m	44243	41	8	11m37s
	P-BwTree	2	0	1	0	1	0	**4**	32945	5333	1h24m	38572	4826	80	1h26m
	P-CLHT	1	0	7	0	1	0	**9**	1580	364	1h23m	10370	476	3	27m27s
	P-CLHT-Aga	3	0	10	0	1	0	**14(1)**	8090	1084	55m49s	39918	248	5	1h43m
	P-CLHT-Aga-TX	2	0	10	4	1	2	**19(17)**	4358	477	2h	27949	242	5	49m25s
	P-Hot	3	0	0	0	4	0	**7**	20132	16403	5h10m	96295	905	155	21m35s
	P-Masstree	0	1	5	0	1	0	**7**	16139	2983	48m27s	115590	142	10	25m6s
PMDK	B-Tree	0	1	0	0	0	5	**6(6)**	1148	131	1h4m	114161	23255	46	20m15s
	C-Tree	0	0	0	0	0	0	**0**	9757	705	2h20m	30113	0	0	20m38s
	RB-Tree	0	1	0	0	0	0	**1(1)**	15342	726	1h24m	376891	5976	64	1h12s
	RB-Tree-Aga	0	2	0	0	0	13	**15(15)**	16188	725	1h23m	386252	82801	219	2h29m
	Hashmap-TX	1	0	0	0	0	0	**1**	8991	802	2h	30364	469	11	21m33s
	Hashmap-atomic	0	2	0	0	0	0	**2(1)**	7931	1078	2h	30068	272	8	1h22m
Server	Memcached	1	0	29	1	0	0	**31(12)**	11089	2746	1h12m	11348	0	0	1h29m
	Redis	0	0	0	0	0	0	**0**	7787	1270	6h49m	260526	0	0	3h3m
Total		**25(3)**	**22(8)**	**102(17)**	**21(8)**	**15(2)**	**20(18)**	**205(56)**	639455	47694	36h37m	1834683	212681	765	18h58m

C-O: persistence order correctness bug **C-A**: persistence atomicity correctness bug **(#)**: number of known bugs
P-U: unpersisted performance bug **P-EFL**: extra flush performance bug **P-EFE**: extra fence performance bug **P-EL**: extra logging performance bug

Table 5. The tested NVM programs, the number of detected bugs, and the detailed statistics of Witcher bug finding.

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" [8]. 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 1(b)) were fixed by reordering source codes [5, 8].

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 [42]. (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* [19], 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.

7.3 Detected Crash Performance Bugs

Table 5 shows that Witcher detected 158 performance bugs from PMDK library and tested applications in total and 113 of them are new bugs. Witcher detected 102 unpersisted performance bugs; 21 extra flush performance bugs; 15 extra fence performance bugs; and 20 extra logging performance bugs, as classified in §2.2.

7.4 Statistics of Witcher Bug Finding

Table 5 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.

For correctness bugs, Witcher finally generated 765 bug reports clustered by operation type and execution path (§4.4). 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

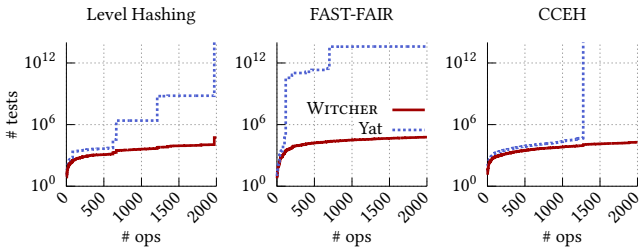

Figure 4. Test space comparison for 2,000 random operations.

same root causes, and we reported and confirmed 25 persistence ordering bugs and 22 persistence atomicity bugs. For performance bugs, Witcher provides execution traces and locations of unpersist stores and extra flush/fence/logging. Root cause analysis of performance bugs is much simpler since it does not require crash simulation.

Table 5 reports testing time. Inferring likely-correctness 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-correctness conditions. We make the comparison in the following sections.

7.5 Scalability and Comparison with Yat

This section evaluates how effectively our likely-correctness condition-based approach can prune the testing space, and thus improve scalability. First, we simulate the existing exhaustive-testing-based tool Yat [52] and compare the number of crash states that Yat will validate using the same trace with 2,000 random operations. Figure 4 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-correctness conditions, significantly reducing the number of test cases (yet detecting many bugs).

Second, Table 5 shows that with likely-correctness 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-correctness 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-correctness conditions are sound. However, the result shows that random pruning does not work, and our approach effectively detects many bugs.

7.6 Bug Detection Effectiveness Comparison

We compared the correctness and performance 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 §7.2 and §7.3, we reported that Witcher discovered 36 new correctness and 113 new performance 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.

For correctness bugs, Witcher detected all seven bugs detected by Agamotto. Agamotto missed two bugs (Bug IDs 1 and 46) due to the lack of application-specific oracles, showing the benefits of output equivalence checking. For performance bugs, both discovered 61 bugs in common. Witcher detected 9 unique bugs and Agamoto found 43 bugs. Recall that the performance bug detection depends on tested program paths. The result implies that Witcher and Agamotto explored different program paths, showing the pros and cons of (guided) symbolic execution by Agamotto and trace-based approach by Witcher. 43 Agamotto-unique performance bugs were found in PMDK libraries that Agamotto's symbolic execution did explore but Witcher did not.

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 performance bugs, Witcher detects the one bug that PMTest detected. XFDetector does not detect new performance bugs. 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` [9], 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 correctness bugs and identify new correctness bugs as well. Witcher uses application-agnostic rules to find application-specific correctness bugs. Witcher detects a new group of application-specific correctness bugs, which cannot be detected by previous works because of the lack of application-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.

7.7 Testing Non-Key-value Store NVM Programs

We extended Witcher for testing a persistent array [6] and a persistent queue [7] 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 [3] in the persistent array.

8 Related Work

Likely-correctness conditions. Prior works have used a concept of likely-correctness conditions to detect program bugs [30, 47, 50, 59, 62, 87], to verify the network [58], and to identify resource leaks [80]. To the best of our knowledge, Witcher is the first work that infers likely-correctness conditions in the context of NVM crash consistency testing.

Output equivalence checking. Burckhardt *et al.* [18] and Pradel *et al.* [72] detect thread-safety violations by comparing the concurrent execution to linearizable executions of a test. Witcher shares a similar idea in the sense that they all compare an observed execution with "oracles", but is uniquely designed to detect NVM crash consistency bugs.

Heuristic-based test space pruning. An iterative context bound [64] or delay bound [29] has been used to (unsoundly yet effectively) prune the thread-interleaving test space when testing multithreaded programs. Witcher uses likely-correctness conditions to prune the NVM state space.

Crash consistency testing in file systems. There has been a long line of research in testing and guaranteeing crash consistency in file systems [20, 21, 33, 48, 63, 75, 79, 84–86]. In-situ model checking approaches such as EXPLODE [85] and FiSC [86] systematically test every legal action of a file system. B3 [63] performs exhaustive testing within a bounded space, which is heuristically decided based on the bug study of real file systems. In contrast, Witcher reduces test space by using inferred likely-correctness conditions. Feedback-driven File system fuzzers, such as Janus [84] and Hydra [48], mutate both disk images and file operations to thoroughly explore file system states.

Crash consistency testing in NVM. Most closely related bug detection works have been discussed in §3.3. In addition, PMFuzz [55] proposes a fuzzing technique to generate diverse inputs for dynamic NVM bug detectors. These inputs can be fed into Witcher (instead of using random test cases). Hippocrates [66] proposes an automated NVM bug fixing solution, placing flush and fence instructions at the right (optimal) positions.

9 Conclusion

We present Witcher, a systematic crash consistency testing framework for NVM-backed persisted key-value stores. Witcher infers *likely-correctness conditions* and performs *output equivalence checking* to validate their violations. This approach allows Witcher to use application-agnostic rules to find application-specific correctness bugs without manual annotations, user-provided consistency checker, or exhaustive testing. Witcher also detects performance bugs during NVM state simulation.

Acknowledgments

We thank the anonymous reviewers and Donald Porter (our shepherd) for their insightful comments and feedback. We thank Sam H. Noh, Vijay Chidambaram, Beomseok Nam, Yu Hua, Sekwon Lee, Hokeun Cha, Pengfei Zuo, and Intel PMDK developers for the bug confirmation. We thank Samira Khan, Baris Kasikci, Brian Demsky, Dong Li, Sihang Liu, Ian Neal, Hamed Gorjiara, and Bang Di for their help in understanding and using their bug detectors. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (No. 2014-3-00035) and in part by National Science Foundation grant No. CSR-2029720.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

[17] 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. ACM.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] Anthony Demeri, Wook-Hee Kim, Madhava Krishnan Ramanathan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. Poseidon: Safe, Fast and Scalable Persistent Memory Allocator. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, pages 207–220. ACM, 2020.

[26] 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.

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

[28] 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.

[29] 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 '11, page 411–422, New York, NY, USA, 2011. Association for Computing Machinery. URL: https://doi.org/10.1145/1926385.1926432, doi:10.1145/1926385.1926432.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. pages 389–400, September 2014.

[36] 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.

[37] 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.

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

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

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

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

[42] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2019. https://software.intel.com/en-us/articles/intel-sdm.

[43] 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.

[44] 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.

[45] 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.

[46] 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*, pages 494–508, 2019.

[47] 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. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. URL: https://doi.org/10.1145/3133907, doi:10.1145/3133907.

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

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

[50] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176, 2006.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 499–512, Oakland, CA, May 2015. USENIX Association. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes.

[59] 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 Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 103–116, New York, NY, USA, 2007. Association for Computing Machinery. URL: https://doi.org/10.1145/1294261.1294272, doi:10.1145/1294261.1294272.

[60] 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.

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

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

[63] 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.

[64] 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 '07, page 446–455, New York, NY, USA, 2007. Association for Computing Machinery. URL: https://doi.org/10.1145/1250734.1250785, doi:10.1145/1250734.1250785.

[65] 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.

[66] Ian Neal, Andrew Quinn, and Baris Kasikci. Hippocrates: Healing persistent memory bugs without doing any harm extended abstract. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

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

[68] Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. volume 19, pages 177–184. ACM, 1984.

[69] Ismail Oukid, Daniel Booss, Adrien Lespinasse, and Wolfgang Lehner. On Testing Persistent-memory-based Software. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 5:1–5:7, San Francisco, California, June 2016.

[70] 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.

[71] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA, June 2016.

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

[73] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of armv8 and transactional models. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. URL: https://doi.org/10.1145/3360561, `doi:10.1145/3360561`.

[74] 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 *ATC '21: 2021 USENIX Annual Technical Conference, July 14-16, 2021*, pages 773–787. USENIX Association, 2021.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 535–544, 2010.

[81] 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.

[82] 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.

[83] 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.

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

[85] 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.

[86] 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.

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

[88] 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.