Debugging and Programming Support for Non-volatile Memory Software

Xinwei Fu

Preliminary Exam

Doctor of Philosophy

in

Computer Science and Application

Ali R. Butt, Virginia Tech, Co-chair

Changwoo Min, Virginia Tech, Co-chair

Dongyoon Lee, Stony Brook University

Danfeng Yao, Virginia Tech

Xun Jian, Virginia Tech

May 10, 2021

Blacksburg, Virginia

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

Debugging and Programming Support for Non-volatile Memory Software

Xinwei Fu

(ABSTRACT)

The advent of non-volatile main memory (NVM) enables the development of crash-consistent software without paying storage stack overhead. NVMs can be accessed by using regular load/store instructions, and programs can recover a consistent state from a persistent NVM state in the event of a software crash, or a sudden power loss. However, NVM programming is hard. Incorrect NVM program leads to serious consequences, *i.e.*, data inconsistency and data loss. Furthermore, NVM bugs are hard to find because: (1) with the presence of a volatile cache, the updates to different NVM locations may not be persisted in the same order as the program (store) order, which leads to an extremely large space of NVM states to explore; (2) traditional mature debuggers, *i.e.gdb*, doesn't take into consider crash consistency. My dissertation question is: *How to make NVM programming easy?* My research goal is to provide debugging and programming support for NVM software, by designing and implementing an automatic, scalable and precise crash-consistency bug detector for both single-thread and multi-threaded NVM software. The long-term vision of this research it to help our community to build bug-free NVM software.

Chapter 1

Introduction

The use of non-volatile main memory (NVM) technologies, such as recently commercialized Intel's Optane DC Persistent Memory [12, 54], is on the rise: *e.g.*, Google Cloud [3] 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 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 program. NVM data on a processor's 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. Moreover, the existing ISA does not provide an instruction to update 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 a 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.

Crash consistency testing tools, especially the ones designed to detect correctness bugs in NVM programs, need to inspect a large number of NVM states feasible on a crash — a new dimension of the testing space. Traditional (non-NVM) testing tools explore two dimensional test space of a program input (*e.g.*, symbolic execution [16, 44] and fuzzing [55, 82]) and/or a thread schedule (*e.g.*, CHESS [59]). As the third dimension, an automatic crash consistency testing tool should be able to enumerate all possible NVM states of a program (as a program may crash at any program point) and check if each NVM state is consistent or not.

Recently, several solutions have been proposed to detect bugs in NVM programs but they have three critical limitations. First, automatic NVM testing tools (*e.g.*, Yat [46], PMReorder [35], and SCMTest [63]) attempt to exhaustively test all possible NVM states (without pruning a test space), resulting in a scalability issue. Second, for a test oracle, all existing NVM testing tools require users to provide a manually-designed, application-specific, full consistency checker to validate the program (*e.g.*, manual source code annotation in PMTest [49] and XFDetector [50], user-provided consistency checker in Yat [46], application-specific bug oracles in Agamotto [61]). Besides required manual effort, it potentially results in both false negatives (missing oracle/annotation) and false positives (incorrect oracle/annotation) depending on the quality of user-provided oracles. Lastly, some testing tools (*e.g.*, PMTest [49], XFDetector [50], and Agamotto [61]) does not automatically explore NVM state test space.

My dissertation question is: *How to make NVM programming easy?* My research goal

[1] 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.

is to provide debugging and programming support for NVM software, by designing and implementing an automatic, scalable and precise crash-consistency bug detector, **NVMTest**, for both single-thread and multi-threaded NVM software. The long-term vision of this research it to help our community to build bug-free NVM software. To achieve the stated goal of this research, we propose two interrelated research thrusts:

1. **Detecting crash consistency bugs in single-thread NVM software**

   To address test space challenge, **NVMTest** infers a set of *likely program invariants* (hereafter invariants for short) that are believed true to be crash-consistent by analyzing source codes and execution traces. **NVMTest** then tests only those NVM states that violate an invariant, instead of relying on exhaustive testing or user's manual annotation. To mitigate the test oracle problem, **NVMTest** takes a novel *metamorphic testing* approach for crash consistency validation. Metamorphic testing [71] fits well in the context of crash consistency of NVM programs as there are strong relations between the outputs of test cases with and without a crash. If a program resuming from an invariant-violating NVM state produces an output different from the executions without a crash, then we can confidently conclude that the program is not crash-consistent, and the invariant violation is indeed a true crash consistency bug.

2. **Detecting crash consistency bugs in multi-threaded NVM software**

   Detecting crash consistency bugs in multi-thread NVM software is challenging, since we need to explore at least two dimensions of test space: (1) NVM state space and (2) thread scheduling space. Our key observation is that multi-threaded NVM bugs only happens when using an volatile value from another thread. To prevent it, lock-based and lock-free fixes can be applied. Both of the fixes guarantee that volatile values must be persisted before another thread using it. Having this in mind, we propose our solution to use single-thread trace searching for NVM racy accesses which

violates lock-based or lock-free fix pattern and then validate is in multi-threaded trace searching. After getting the multi-threaded trace, a simulated crashed NVM image can be constructed based on it and the metamorphic testing can be applied to check whether there is bug.

Proposal Structure. The rest of this Ph.D. preliminary proposal is organized as follows: chapter 2 introduces the background the this research; chapter 3 compares our research against the related works; chapter 4 discusses how we detect single-thread NVM bugs; chapter 5 discusses the proposal for detecting multi-threaded NVM bugs; chapter 6 draws our conclusion.

Chapter 2

Background

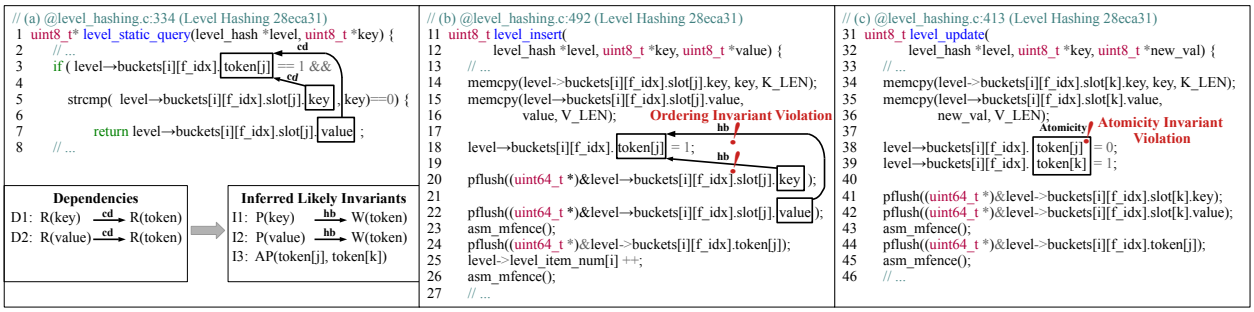
This section demonstrates the types of correctness bugs (section 2.1), performance bugs (section 2.2) and multi-threaded bugs (section 2.3) that **NVMTest** found in crash-consistant NVM programs, along with real-world examples.

2.1 Correctness Bugs

In order to understand the property of crash consistency bugs, we investigated the design, implementation, and issues in real-world open source NVM programs, including Intel's PMDK libraries, NVM-optimized key-value indexed, and NVM databases. We found that persistent ordering and persistent atomicity violations are common root cause in many crash consistency bugs and application-specific knowledge is necessary to identify such violations.

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 of 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



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

Figure 2.1: Using the likely invariants inferred from (a), **NVMTEST** detects two correctness bugs (b) and (c) in Level Hashing.

line flush and a store fence instructions when updating multiple NVM locations.

For example, Level Hashing [83] introduces *log-free* write operations. It maintains a flag token for each key-value slot where token 0 denotes the corresponding key-value slot is empty and token 1 denotes non-empty. Figure 2.1 (b) shows the log-free `level_insert` function. It intends to update the key-value slot (Lines 14, 15) before updating 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 Line 20), 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 4.2(h)), implying that the insert operation failed to provide an atomic (all or nothing) semantic upon a crash. The persistent barrier 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 2.1(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 the 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 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.

2.2 Performance Bugs

Previous studies [49, 61] found that performance bugs – sub-optimal use of NVM – 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. We classify NVM performance bugs in the following four types similar to prior work:

(1) Unpersisted performance bugs We found that 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 transactional programming library (*e.g.*, Intel's PMDK) for crash consistency, the NVM data should be

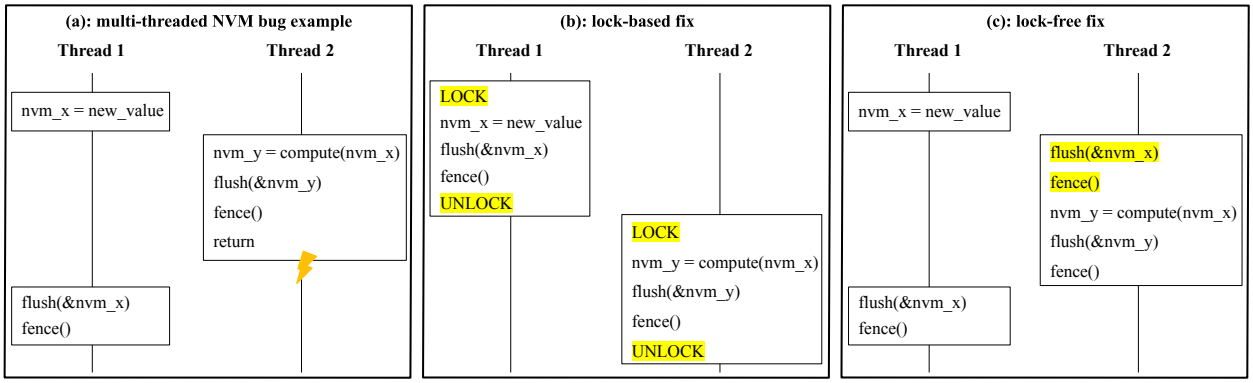


Figure 2.2: An example of multi-threaded NVM bug and its potential fixes.

(undo) logged before it is modified first time. Logging the same NVM region redundantly within a transaction will unnecessarily degrade performance.

2.3 Multi-threaded Bugs

Figure 2.2 (a) gives an example of multi-threaded NVM bug. There are two threads running concurrently. Thread 1 writes `new_value` to an NVM location `nvm_x`, and then persists it by using `flush` and `fence`. Thread 2 reads `nvm_x` and writes to `nvm_y` based on the computation on `nvm_x`, then persists `nvm_y`. An inconsistency could happen if: (1) Thread 2 preempts after `nvm_x` is assigned but before `nvm_x` gets persistent; (2) a crash after `nvm_y` is persisted but `nvm_x` is not persisted. In this case, after the crash, the NVM image doesn't have the updated value of `nvm_x` but it does have updated value of `nvm_y` which is computed based on `new_value`.

There are two potential fixes for this example. One potential fix is lock-based fix as shown in Figure 2.2 (b). It adds locks to prevent Thread 2 to preempt. The other potential fix is lock-free fix as shown in Figure 2.2 (c). It persists `nvm_x` by adding `flush` and `fence` before using `nvm_x` for `nvm_y` computation. Both of the fixes guarantee that `nvm_x` must be persisted before using it.

Chapter 3

Related Work

Crash consistency testing in NVM software Table 3.1 summarizes how **NVMTest** is different from existing crash consistency testing tools when it comes to detecting correctness bugs. For performance bugs, **NVMTest** is similar to existing work, and subsection 4.4.6 later shows the pros and cons of **NVMTest**'s trace-based approach, compared to symbolic execution-based approach in Agamotto [61].

Exhaustive testing tools such as Yat [46], PMReorder [35], and SCMTest [63] attempts to permute all possible NVM states on a crash. However, they often do not scale. For example, testing Level Hashing with 2000 (random) operations, the Yat tests 10^{31} total permutations (see subsection 4.4.5). 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 [40]. Later in subsection 4.4.5, we show that **NVMTest** effectively prune NVM state test space using invariant-based approach.

The test space explosion problem motivated the *annotation-based approach*, such as PMTest [49] and XFDetector [50]. They however puts a significant annotation burden on the developers, raising soundness, completeness, and scalability concerns. A missing/wrong annotation may lead to false negatives/positives. Annotating a large NVM software soundly and precisely is very challenging. Note that PMTest lacks support for detecting persistence atomicity violations such as Figure 2.1(c). XFDetector relies on user's manual investigation for validation. Agamotto [61] takes a different approach to use symbolic execution to explore input

	Test space exploration		Crash consistency validation (oracle)
	Input	**NVM State**	
Yat [46] PMReorder [35]	user-provided test case	exhaustive	user-provided oracle
PMTest [49] XFDetector [50]	user-provided test case	manual annotation	user-provided oracle
Agamotto [61]	symbolic execution	user-provided oracle	user-provided oracle
NVMTEST (this work)	user-provided test case	systematic with invariant-based pruning	automatic metamorphic testing

Table 3.1: Comparision with existing crash consistency testing tools.

test space (program paths) and provides universal bug oracles for common bug patterns (*i.e.*, missing or redundant flush/fence bug patterns). However, for app-specific correctness bugs, Agamotto still requires users to provide test oracles. Later in subsection 4.4.6, we show that **NVMTEST** can detect more bugs than the prior testing tools using invariant- and metamorphic-testing based approach without manual annotations.

Likely-invariants based testing A concept of likely-invariants has been used to detect program bugs [26, 42, 45, 52, 56, 81], to verify the network [51], and to detect resource leak [74]. Notably, Engler *et al.*'s version (called beliefs) [26] enables automatic analysis of likely correctness conditions without in-depth knowledge. To the best of our knowledge, **NVMTEST** is the first work that infers likely invariant in the context of crash consistency testing for NVM programs.

Output-equivalence based testing Burckhardt *et al.* [15] and Pradel *et al.* [66] detect thread-safety violations, comparing the concurrent execution to linearizable executions of a test. **NVMTEST**'s metamorphic testing shares a similar idea of these two works in the sense that they all compare an observed execution with "oracles", but is uniquely designed to detect for NVM crash consistency bugs.

Crash consistency testing in file systems There has been a long line of research in testing and guaranteeing crash consistency in file systems [18, 19, 28, 43, 57, 68, 73, 78, 79, 80]. In-

situ model checking approaches such as EXPLODE [80] and FiSC [79] systematically test every legal action of a file system with minimal modification. B3 [57] performs exhaustive testing within a bounded space, which is heuristically decided based on the bug study of real file systems. **NVMTest** reduces test space by using inferred invariants, unlike limiting testing space in B3. Feedback-driven File system fuzzers, such as Janus [78] and Hydra [43], mutate both disk images and file operations to thoroughly explore file system states. We believe **NVMTest**'s test case generation can be further improved by adopting feedback-driven fuzzing techniques.

Chapter 4

Detecting crash consistency bugs in single-thread NVM software

4.1 Overview of Our Approach

We first introduce how NVMTest finds correctness bugs (subsection 4.1.1) and performance bugs (subsection 4.1.2).

4.1.1 Correctness Bug Finding

To detect correctness bugs, NVMTest infers likely invariants and uses metamorphic testing to validate the crash consistency of invariant-violating NVM states.

Inference of Likely Invariants

We propose a novel *likely invariant-based* approach to detect NVM crash consistency bugs scalably and automatically. Our key observation is that programmers often left some hints on what they want to ensure in the source codes. Thus we can infer a set of likely-correctness conditions, which we refer to as *likely invariants* (hereafter invariants for short), by analyzing source codes and execution traces.

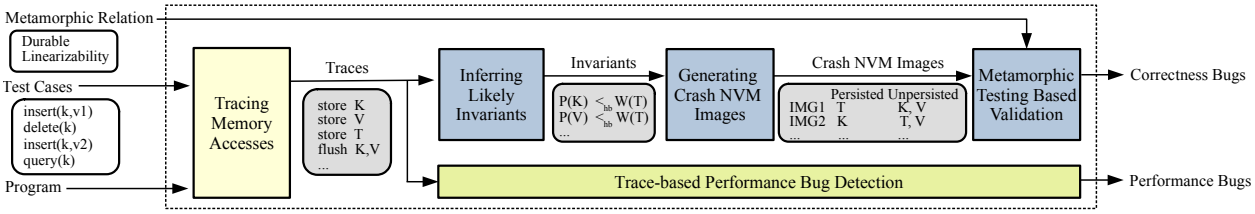


Figure 4.1: The architecture of NVMTest. NVMTest automatically detects (application-specific) correctness bugs (in blue) and performance bugs (in green) based on a given test case and its trace (in gray) without no manual annotation/oracle nor exhaustive testing.

Using the Level Hashing example as mentioned earlier, let us demonstrate how we can infer an invariant from the query (or lookup) function `level_static_query` in Figure 2.1(a), and apply it to find the correctness bugs in `level_insert` and `level_update` in Figure 2.1(b) and (c). `level_static_query` reads the key/value only if the token is non-empty. In other words, there is a 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)` $\xrightarrow{cd}$ `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 [8, 53, 76], logging implementations [13, 30, 31, 38, 75], persistent data structures [20, 22, 32, 47, 48, 60, 64], memory allocators [14, 34, 65, 70], and file systems [21, 23, 24, 41, 77]. 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 2.1(b)). (2) The reader checks if the flag is set before reading the key and value, which we call *"guarded read"* (Figure 2.1(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 2.1(a), we infer the first *persistence ordering invari-*

ant: a key-value pair should be persisted before a token (*i.e.*, `P(slot[j].key/value)` $\xrightarrow{\text{hb}}$ `W(token[j])`). We then extend it to the second *persistence atomicity invariant*: the updates of two or more guardians should be atomic. Otherwise, an atomic update of multiple key-value slots cannot be guaranteed (*i.e.*, `AP(token[j],token[k])`).

Later we find that `level_insert` violates the persistence ordering invariant at Line 18, and `level_update` violates the persistence atomicity invariant at Line 39. NVMTEST tests only NVM states that violate the inferred invariants. 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, we can significantly reduce testing space without the developer's manual annotation. As an iterative context bound [58] or delay bound [25] has been used for systematic testing of multithreaded program, NVMTEST uses likely-invariants to avoid the NVM state test space explosion. In subsection 4.2.2, we present more generalized rules to infer likely invariants beyond guarded protection. NVMTEST does not require prior knowledge of truth and does not assume invariants are always correct: if two invariants contradict, we test both cases to discern which one is correct using metamorphic testing.

Validation with Metamorphic Testing

We propose to use *metamorphic testing* to validate if an NVM state that violates an inferred invariant is indeed inconsistent, indicating a crash consistency bug. The idea behind metamorphic testing [71] is that even if we do not know the correct output of a single test case, we might still know the relations between the outputs of multiple test cases, using domain-specific knowledge. We can check the software for these *metamorphic relations*: *e.g.*,

for a sine function `sin()`, we can test `sin(π-x)=sin(x)`. If they don't hold, it is a sure sign that the software has some defects.

Our key insight is that metamorphic testing fits well in the context of crash consistency of NVM programs as there are strong relations between the outputs of test cases with and without a crash. Many NVM programs including a persistent key-value store aims to provide durable linearizability [39] at the operation granularity. 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, **NVMTest** can validate crash consistency by comparing the outputs of executions with and without a crash. If a program recovering from an invariant-violating NVM state (after a crash) 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 invariant violation is a true bug.

Using durable linearizability metamorphic relation allows **NVMTest** to automatically detect correctness bugs without manual annotations or user-provided full consistency checker. Metamorphic testing at the same time requires that the test case is deterministic: *i.e.*, given the same input, a program should produce the same output. Moreover, metamorphic testing rely 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. Besides durable linearizability, users may consider using other metamorphic relations: *e.g.*, buffered durable linearizability [39], and strict serializability for transactional programs [67].

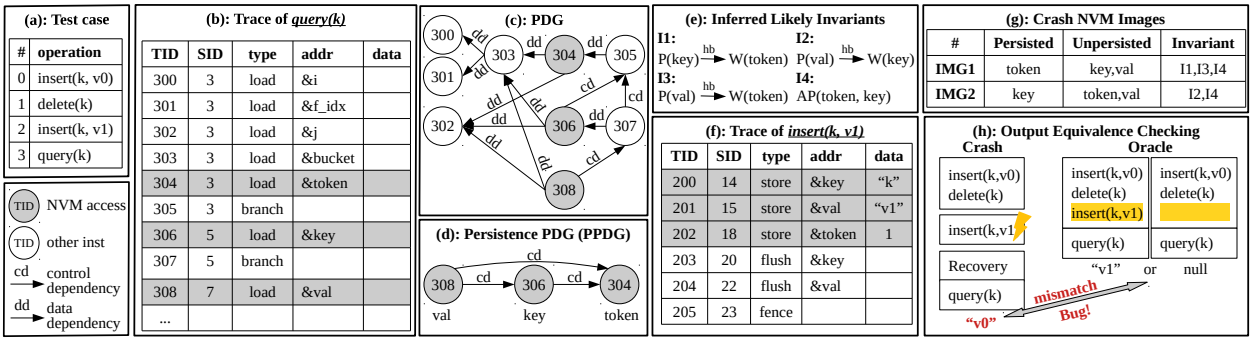


Figure 4.2: An example of NVMTEST's correctness bug detection steps.

4.1.2 Dynamic Trace Based Performance Bug Finding

NVMTEST 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, as used in [49, 61]. For example, to detect an extra flush performance bug, the persistence state of the cacheline to be flushed before executing the flush instruction is needed. NVMTEST leverages the collected dynamic program trace and detects performance bugs during NVM persistence simulation.

4.2 Design of NVMTEST

This section describes the detailed design of NVMTEST. Figure 4.1 illustrates NVMTEST architecture that takes as input (1) a target NVM program (2) a test case and (3) metamorphic relation and reports as output detected correctness and performance bugs in the program. NVMTEST first instruments the program and runs the test case to collect a memory trace (subsection 4.2.1). For correctness bugs, NVMTEST infers invariants from the trace (subsection 4.2.2), constructs a set of crash NVM images violating the invariants (subsection 4.2.3), and performs metamorphic testing to validate if the invariant violation is a true correctness

bug (subsection 4.2.4). NVMTest analyzes the same trace to detect performance bugs as well (subsection 4.2.5).

4.2.1 Tracing Memory Accesses

NVMTest instruments an NVM program using an LLVM compiler pass [5] and executes the instrumented binary with a test case to collect the execution trace. We trace load, store (including the updated value), branch, call/return, flush and memory fence instructions. We order the instructions in a trace by protecting our tracing code using a global mutex so we can easily analyze traces of multi-threaded programs.

Suppose we trace Level Hashing in Figure 2.1 using the test case with four operations in Figure 4.2(a). Figure 4.2(b) shows the trace of the last `last_static_query(k)`. Each trace includes a unique Trace ID (TID), a Static instruction ID (SID), which is essentially the instruction location in the binary, and instruction type. For `load` and `store`, NVMTest additionally traces its address, length (not shown), and data (for `store`), and whether it accesses DRAM (white) or NVM (gray).

4.2.2 Inferring Likely Invariants

We first describe a set of inference rules for (1) persistence ordering invariants and (2) persistence atomicity invariants. Then we explain how NVMTest uses program dependence analysis to apply the rules and infer the corresponding invariants from the trace.

Invariant 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. **NVMTEST** then infers Persistence Ordering (PO) likely invariants that "`X` should be persisted before `Y`" or Persistence Atomicity (PA) invariants that "`X` and `Y` should be persisted atomically". From the invariant, **NVMTEST** later constructs an NVM state that violates it – *e.g.*, "`Y` is persisted, but `X` is not" (subsection 4.2.3) and tests if the invariant violation is a true crash consistency bug using metamorphic testing (subsection 4.2.4). In other words, for two NVM addresses `X` and `Y`, if **NVMTEST** does not detect any dependency, it does not test such cases involving `X` and `Y` and saves the test time (assuming that independent NVM objects do not lead to an inconsistent state). Table 4.1 summarizes the inference rules:

(PO1) A data dependency implies a persistence ordering. Consider the code "`Y=X+1`" where the write of `Y` is data-dependent on the read of `X` (which we denote $\texttt{W(Y)} \xrightarrow{\text{dd}} \texttt{R(X)}$). From the data dependency, we infer a PO invariant that for another code region where `X` and `Y` are updated, the developer would want `X` to be *persisted before* updating `Y` (*i.e.*, $\texttt{P(X)} \xrightarrow{\text{hb}} \texttt{W(Y)}$ where $\xrightarrow{\text{hb}}$ stands for happens-before). Otherwise, she may update `Y` based on "unpersisted" `X`. Based on the reasoning, `PO1` in `PO1` in Table 4.1 says: for two NVM locations `X` and `Y`, if we find a Hint $\texttt{W(Y)} \xrightarrow{\text{dd}} \texttt{R(X)}$, we infer a Likely Invariant $\texttt{P(X)} \xrightarrow{\text{hb}} \texttt{W(Y)}$. We later test an invariant-violating NVRAM state in which `Y` is persisted, but `X` is not.

(PO2) A control dependency implies a persistence ordering. Based on the same rationale, we infer a PO invariant 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 $\texttt{W(Y)} \xrightarrow{\text{cd}} \texttt{R(X)}$, we infer a Likely Invariant $\texttt{P(X)} \xrightarrow{\text{hb}} \texttt{W(Y)}$. From the invariant, we test a state where only `Y` is persisted.

(PO3) A guarded read implies a persistence ordering. As discussed in section 4.1.1, 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 invariant at a writer side. In other words, `PO3` says: for two NVM locations `X` and `Y`, if we find a Hint `R(Y)` $\xrightarrow{\text{cd}}$ `R(X)`, we infer a Likely Invariant `P(Y)` $\xrightarrow{\text{hb}}$ `W(X)`. We then validate an invariant-violating NVM state such that `X` is persisted but `Y` is not. Note that here `X` is a guardian in the guarded read pattern (*e.g.*, `token` in Figure 2.1) and thus it should be persisted last (after `key` and `value`).

(PA1) Guardian implies persistence atomicity. As in the `PO3` ordering invariant, we can find a set of guardians: *e.g.*, `token[j]` and `token[k]` in Figure 2.1. A program state could be inconsistent if they are not updated atomically — none guards the guardians. Based on this observation, we infer an PA invariant such that two or more guardians should be atomically updated. `PA1` says: for two guardians `X` and `Y` from `PO3`, we infer the Likely Invariant `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.

Program Analysis for Invariant Inference

NVMTest performs program dependence analysis to infer likely invariant from the source codes and execution traces. NVMTest first constructs Program Dependence Graph (PDG) [27, 29, 62] where a node represents a traced instruction, and an edge represents data or control dependency. Then, NVMTest simplifies the PDG into what we called Persistence Program

#	Hint		Likely Invariant		NVM Image	
	Example	Rule	Example	Rule	$\mathbb{P}$	$\mathbb{U}$
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` $\mathbb{P}$: persisted $\mathbb{U}$: 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 4.1: Three inference rules `PO1`–`PO3` for persistence ordering invariants and one rule `PA1` for atomicity invariants.

Dependence Graph (PPDG) that captures dependencies between NVM accesses to make it easy to apply the invariant inference rules. For example, Figure 4.2(c) shows the PDG of the trace (b), and (d) shows the PPDG.

NVMTEST uses a mix of static and dynamic trace analysis to construct a PDG. When instrumenting the source code for tracing (subsection 4.2.1), it performs static analysis to capture register-level data and control dependency. Then it extracts memory-level data dependence by analyzing memory-level data-flow in the collected trace. This dynamic memory-level data dependency analysis improves PDG's precision compared to static-only analysis which suffers from the imprecision of pointer analysis. The static instruction IDs (binary address) are used to map static and dynamic information.

NVMTEST converts a PDG to a PPDG as follows. Initially, the PPDG has only (gray) NVM nodes. NVMTEST 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, **NVMTest** then applies the inference rules in Table 4.1 to infer likely invariants. For each edge and two nodes in the PPDG, **NVMTest** considers the type of edge (control vs. data) and the type of instructions (`store` vs. `load`). When **NVMTest** finds a Hint, it records the corresponding Likely Invariant. For example, the PPDG in Figure 4.2(d) shows that `R(key)` $\xrightarrow{\text{cd}}$ `R(token)`. Based on `PO3`, we infer the invariant `I1: P(key)` $\xrightarrow{\text{hb}}$ `W(token)` in (e). Similarly, we can infer the persistence ordering invariants `I2` and `I3`. Moreover, as `token` and `key` are guardians for guarded reads, based on `PA1`, we infer the persistence atomicity invariant `I4: AP(token,key)`.

4.2.3 Generating Crash NVM Images

The next step is to generate a set of crash NVM images[1] that violate the invariants. Later in subsection 4.2.4, we will describe how **NVMTest** loads these NVM images and uses metamorphic testing to validate if an invariant violation is a true bug or not.

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

Simulating Cache and NVM States

The goal of the cache/NVM simulations is to generate only feasible NVM states that violate likely invariants 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,

[1] In PMDK, an NVM image is a regular file containing an NVM heap state created, loaded, and closed by PMDK APIs [36].

NVMTEST simulates the effects of `store`, `flush`, and `fence` instructions along the trace while honoring the memory (consistency) model of a processor. In particular, NVMTEST supports Intel's x86-64 architecture model, as in Yat [46]. 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 [37, 72]).

Consider the trace of Level Hashing's `level_insert` code in Figure 4.2(f). After simulating the first three `store` instructions (TID 200-202), there could be multiple valid cache/NVM states. For example, the data ``k'' for `key` could either remain in a cache (unpersisted) or could be evicted (persisted). The same is true for the `val` and `token`. However, after finishing the execution of the last `fence` instruction (TID 205), `key` and `val` are guaranteed to be persisted (due to `flush` and `fence`). Still, `token` could be either unpersisted or persisted.

Detecting Invariants Violations

During the simulation, NVMTEST checks if there could be an NVM state that violates a likely invariant before executing each `fence` instruction because the `fence` ensures a persistent state change. NVMTEST 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 4.2(f) again. Before we execute the last `fence` instruction (TID 205), we check the four invariants against the trace as shown in (e). For instance, `I1` says that `P(key)` $\xrightarrow{\text{hb}}$ `W(token)`. The invariant violating state is the one that `token` is persisted, but `key` is not. We check if this invariant violating case 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 store ID, which specifies where to crash and which store to be persisted, respectively. NVMTest repeats the process along the trace and generates a set of crash NVM images that will be validated in the next step.

4.2.4 Metamorphic Testing Based Consistency Validation

NVMTest validates the invariant-violating crash NVM images and detects crash consistency bugs using metamorphic testing. In particular, NVMTest focuses on testing metamorphic relation for durable linearizability [39]. 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 (rollbacked). Thus, the program resumed from a crash NVM image should produce the same output as one of these two committed or rollbacked executions, which we call *oracles*.

Consider the example in Figure 4.2 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`.

NVMTest 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 rollbacked – 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`.

For metamorphic testing, NVMTEST loads a crash NVM image, runs a recovery code (if 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 not persisted) – NVMTEST 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 NVMTEST does not report it as correctness bugs.

One key benefit of metamorphic testing 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 finding the root causes, NVMTEST 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 for bug fixing because 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.

4.2.5 Performance Bug Detection

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

4.3 Implementation

We built tracing and program dependency analysis based on Giri [69], a dynamic program slicing tool implemented in LLVM [5]. Our Giri modification comprises of around 3600 lines of C++ code. Other NVMTest components are written in 4400 lines of Python code.

Our current prototype supports an NVM program built on PMDK `libpmem` or `libpmemobj` libraries to create/load an NVM image from/to disk. To ensure the virtual address of `mmap`-ed NVM heap the same across different executions, we set `PMEM_MMAP_HINT` environment variable. NVMTest runs PPDG construction, crashed NVM image generation, and output equivalence checking in parallel.

The current prototype does not support kernel-level NVM programs. Tracing a kernel execution and inferring invariants can be supported with more engineering efforts. Systematic testing of a kernel-level program (*e.g.*, using virtualization like Yat) is left as future work.

4.4 Evaluation

This section first discuss evaluation methodology (subsection 4.4.1) and present experimental results: detected correctness bugs (subsection 4.4.2) and performance bugs (subsection 4.4.3) with detailed statistics in likely invariant inference and metamorphic testing (subsection 4.4.4). Moreover, we evaluate the scalability (subsection 4.4.5) and bug detection effectiveness (subsection 4.4.6) of **NVMTest**, compared to prior testing tools.

4.4.1 Evaluation Methodology

Tested NVM Programs We evaluate **NVMTest** with four groups of 20 (in total) real-world NVM programs[2] (Table 4.2). The first group includes five highly optimized persistent key-value indexes, which are backbone of many storage systems. For high performance, they all have their own crash consistency mechanism using *low-level* (LL) persistence primitives such as `flush` and `fence` instructions. For example, FAST-FAIR [32] incorporate 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 [48]. We used three different versions/configurations of P-CLHT to compare with Agamotto. 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 PMDK's six (example) applications. They used PMDK's low-level (LL) or *transactional* (TX) persistence programming model. We used two version of RB-tree for the comparison with Agamotto. The last group includes PMDK-based two server programs `Memcached` and `Redis` using PMDK's LL and TX persistence APIs, respectively. We also note that Memcached and Redis maintain only a part of

[2]We did not test PMFS [33], pmem-redis [7], and NVM Direct [6] that have been tested by prior work as they are not based on PMDK. They are not actively maintained, either.

its application state in NVM as a persistent hash table, which turn 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 introduced additional memory operations, which may potentially affect the bug detection evaluation. For all applications, **NVMTest** traces and analyzes not only application code but also entire PMDK library such as persistence heap allocation and transactional undo logging logics, validating both the PMDK library and NVM programs using them.

Test Cases **NVMTest** requires a deterministic test case such that its output is deterministic for a given input for metamorphic testing (section 4.1.1). Any deterministic test case with a 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 indexes providing well-known key-value interfaces such as `insert`, `delete`, `update`, `query`, and `scan`. **NVMTest** 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 a unused key for `insert`; and (2) to generate a used key for other operations (`delete`, `update`, `query`, and `scan`) which work on existing keys.

We run the NVM programs with a test case consisting of 2000 randomly generated operations. We found that 2000 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

	Application	Version	Lib	Design	Core NVM Construct	Concurrency
NVM KV Index	WOART [47]	5b4cf3e	PMDK v1.8	LL	radix tree	ST
	WORT [47]	5b4cf3e	PMDK v1.8	LL	radix tree	ST
	Fast Fair [32]	c86f5fb	PMDK v1.8	LL	B+ tree	LB
	Level Hash [83]	28eca31	PMDK v1.8	LL	hash table	ST
	CCEH [60]	d53b336	PMDK v1.8	LL	hash table	LB
RECIPE	P-ART [48]	5b4cf3e	PMDK v1.8	LL	radix tree	LB
	P-BwTree [48]	5b4cf3e	PMDK v1.8	LL	B+ tree	LF
	P-CLHT [48]	5b4cf3e	PMDK v1.8	LL	hash table	LB
	P-CLHT-Aga [48]	53923cf	PMDK v1.8	LL	hash table	LB
	P-CLHT-Aga-TX [48]	53923cf	PMDK v1.8	TX	hash table	LB
	P-Hot [48]	5b4cf3e	PMDK v1.8	LL	trie	LB
	P-Masstree [48]	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 4.2: The description of tested NVM programs.

16-core Intel Xeon Gold 5218 processors (2.30GHz), 192 GB DRAM, and 512 GB NVM.

4.4.2 Detected Correctness Bugs

NVMTEST detected 48 (38 new) crash consistency bugs from 18 programs. There were 26 persistence ordering bugs and 22 persistence atomicity bugs. All the bugs were confirmed by the developers. We present the full list of correctness bugs detected by NVMTEST, source code locations, their impacts, and their fix strategies in Table 4.3.

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 4.3) 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 in 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 or performance degradation.

Bug Case Studies Many detected bugs are not shallow. For instance, Bug ID 1 was a persistence ordering bug inside of PMDK's persistent pool/heap allocation function `pmemobj_tx_zalloc`, classified as "Priority 1: showstopper" [2]. 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. Our study reveals that it is hard for a developer to reason about all possible NVM states (as reasoning about all possible thread interleaving is difficult for multithreaded programming).

Fixing persistence ordering bugs NVMTest detected 26 persistence ordering bugs in total. 14 persistence ordering bugs occurred because developers did not add persistence primitives or passed incorrect addresses as parameters. Fixing these bugs is straightforward: adding required persistence primitives (`flush`/`fence`) or passing the correct address. The rest 12 persistence ordering bugs had persistence primitives, but they persist multiple stores in an incorrect order. Fixing them requires reordering persistence primitives. For example, Bug ID 1 in PMDK's persistent pool/heap allocator and Bug ID 7 in `level_insert` (Figure 2.1(b)) were fixed by reordering source codes [2, 4].

Fixing persistence atomicity bugs NVMTest detected 22 persistence atomicity bugs in total. Four cases (Bug IDs 41-44) were a missing logging problem in transactional programs. Fixing is relatively trivial – just add a logging. For rest 18 bugs appeared in low-level NVM programs, all of them indeed required design or implementation-level changes. We observed the following four fixing strategies from the developers: (1) One solution is to merge mul-

tiple writes into *one word-size write* within one cache line to guarantee atomicity [37]. (2) The second solution is 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* [17], where an operation started by one thread but failed is later completed by another thread. (3) The third is to make program *crash-inconsistency-recoverable*. This solution introduces a recovery code which is executed after a crash and fixes any observed inconsistency. (4) The last approach is to use *logging/transaction* techniques.

4.4.3 Detected Crash Performance Bugs

Table 4.4 shows that NVMTEST detected 137 performance bugs (along with correctness bugs) from PMDK library and tested applications in total and 115 of them are new bugs. NVMTEST detected 100 unpersisted performance bugs. Those bugs were due to developers placing volatile data, which is reconstructed after crash, into NVM. NVMTEST detected 21 extra flush performance bugs. Those bugs were mainly due to two reasons: developers added extra flush by mistake especially in different functions; or developers could not figure out whether two memory regions belong to the same cacheline at static time. NVMTEST detected 15 extra fence performance bugs. Developers had misunderstanding of the fence semantics and always put a fence before a flush. NVMTEST detected 1 extra logging performance bugs. Two logging instructions were added in different functions within the same transaction.

4.4.4 Statistics of NVMTEST Bug Finding

Table 4.4 also presents the detailed statistics of each major step in NVMTEST. Across 20 NVM programs, when tested with 2,000 operations, NVMTEST infers in total 639K (32K

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 [2]
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 [4]
	8	✓	level_hashing.c:507	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [4]
	9	✓	level_hashing.c:417	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [4]
	10	✓	level_hashing.c:610	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [4]
	11	✓	level_hashing.c:616	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [4]
	12	✓	level_hashing.c:657	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [4]
	13	✓	level_hashing.c:677	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [4]
	14	✓	level_hashing.c:545	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [4]
	15	✓	level_hashing.c:560	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [4]
	16	✓	level_hashing.c:445	C-O	Incorrect persistence order	Unexpected key-value	persistence reorder [4]
	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 [4]
	23	✓	level_hashing.c:444	C-A	Atomicity between two metadata	Lost key-value	merge to word size [4]
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 [10]
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
	36	×	clht_lb_res.c:583	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
P-CLHT-Aga-TX (3)	34	✓	clht_lb_res.c:177	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
	35	✓	clht_lb_res.c:559	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
	36	×	clht_lb_res.c:583	C-O	Missing persistence primitives	Lost key-value	add persistence primitives
P-HOT (3)	37	✓	TwoEntriesNode.hpp:30	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives [10]
	38	✓	HOTRowexNode.hpp:315	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives [10]
	39	✓	HOTRowex.hpp:270	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives [10]
P-Masstree (1)	40	✓	masstree.h:1378	C-A	Atomicity in node splitting	Inconsistent structure	logging/transaction
B-Tree (1)	41	×	btree_map.c:201	C-A	Missing logging in a transaction	Inconsistent structure	add logging
RB-Tree (1)	42	×	rbtree_map.c:417	C-A	Missing logging in a transaction	Inconsistent structure	add logging
RB-TreeAga (2)	43	×	rbtree_map.c:174	C-A	Missing logging in a transaction	Inconsistent structure	add logging
	44	✓	rbtree_map.c:355	C-A	Missing logging in a transaction	Inconsistent structure	add logging
Hashmap-TX (1)	45	✓	hashmap_tx.c:281	C-O	Use-after-free	Unexpected op failure	copy before free
Hashmap-atomic (2)	46	×	hashmap_atomic.c:129	C-A	Atomicity when creating hashmap	Inconsistent structure	logging/transaction
	47	✓	hashmap_atomic.c:198	C-A	Atomicity when assigin pool id and offset	Inconsistent structure	inconsistency-recoverable design
Memcached(1)	48	×	items.c:538	C-O	Missing persistence primitives	Inconsistent structure	add persistence primitives

C-O: persistence order correctness bug **C-A**: persistence atomicity correctness bug
P-U: unpersisted performance bug **P-EFL**: extra flush performance bug
P-EFE: extra fence performance bug **P-EL**: extra logging performance bug

Table 4.3: List of correctness bugs discovered by **NVMTest**. All 48 bugs have been confirmed by authors or existing tools, and 38 of 48 bugs are new. There are 26 persistence ordering bugs and 22 persistence atomicity bugs. One bug (ID 1) is in the PMDK library.

on average) ordering invariants and 48K (2.4K) atomicity invariants. **NVMTest** generated 1835K (92K) crash NVM images, 213K (11K) of which failed the validation step.

NVMTest finally generated 765 bug reports for correctness bugs. To analyze the root cause of the correctness bugs and to communicate with the developers, we investigated all generated bug reports. **NVMTest** provides sufficient information for root cause analysis including execution trace, crash location, persisted and unpersisted writes, and a crash NVM image,

	Name	Correctness		Performence				Total	Likly Invariant inference			Metamorphic Testing Based Consistency Validation			
		C-O	C-A	P-U	P-EFL	P-EFE	P-EL		# ordering invariants	# atomicity invariants	execution time	# crash NVM images	# image w/ output mismatch	# cluster	execution time
Library	libpmemobj	1	0	3	0	0	0	**4**	-	-	-	-	-	-	-
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	3	0	10	3	1	0	**17(10)**	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	1	**2(2)**	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	0	**2(1)**	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	1	0	0	**3(2)**	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		**26(3)**	**22(7)**	**100(17)**	**21(3)**	**15(1)**	**1(1)**	**185(32)**	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 4.4: The tested NVM programs, the number of detected bugs, and the detailed statistics of NVMTEST bug finding.

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 NVMTEST's reports, manually but guided by `gdb`-based debugging. For those performance bugs, NVMTEST 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.

NVMTEST run testing tasks in parallel and we found that NVMTEST prototype is fast enough for practical use. Invariant inference took a few minutes to seven hours. The validation step took a few minutes to three hours. The validation step is proportional to the number of crash NVM images and the cost of each test run.

4.4.5 Scalability and Comparison with Yat

This section evaluate how effectively our likely invariant-based approach can prune the testing space, and thus improves scalability. First, we simulate the existing exhaustive-testing-based tool Yat [46] and compare the number of crash states that Yat will validate using the

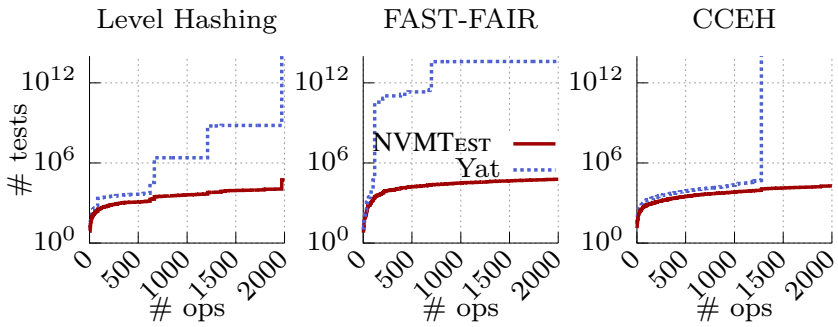
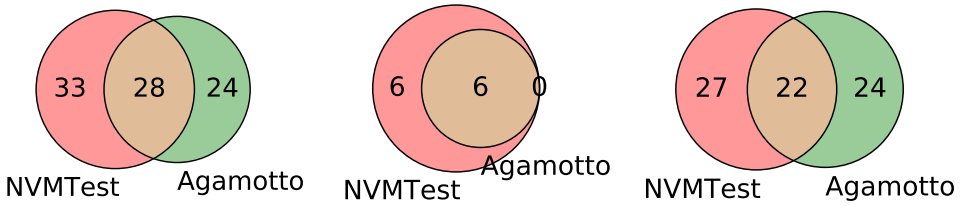


Figure 4.3: Test space comparison for 2000 random operations.

same trace with 2000 random operations. Figure 4.3 shows the representative results for Level Hashing, FAST-FAIR, and CCEH programs. The test space of Yat is several orders larger than **NVMTest**. Sudden spikes happen in Yat when there is a rehashing in Level Hashing and CCEH or a node split/merge in FAST-FAIR. **NVMTest** only tests when there is an invariant violation, significantly reducing the number of test cases (yet detecting many bugs).

Second, Table 4.4 shows that with likely invariants, **NVMTest** 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 **NVMTest** may (unsoundly) miss. However, as shown in Yat simulation, the NVM state space is too huge to explore them all. Alternatively, we tested 10 million randomly chosen NVM states (without considering likely invariants), which is 166x-526x larger NVM test space. The results show that the random 10M case can only detect one or two of the bugs that **NVMTest** detected, yet there was no new bug that **NVMTest** missed. Without a full search, we cannot conclude that likely invariant is sound. However, our random 10M test shows that random pruning does not work, and likely-invariant-based approach effectively detects many bugs.



(a) All bugs (b) Correctness bugs (c) Performance bugs

Figure 4.4: Bug detection comparison between **NVMTest** and Agamotto, using P-CLTH-Aga-TX, 6 apps in PMDK, Memcached and Redis.

4.4.6 Bug Detection Effectiveness Comparison

We compared the correctness and performance bugs detected by **NVMTest** and other existing tools: Agamotto, PMTest, and XFDetector.

Agamotto **NVMTest** and Agamotto tested 9 programs in common: P-CLHT-Aga-TX (in RECIPE), all the six programs in PMDK, Memcached and Redis (v3.2). Note that **NVMTest** test the PMDK library as well. Figure 4.4 shows the comparison result.

For correctness bugs, **NVMTest** detected all 6 bugs detected by Agamotto. **NVMTest** detected 6 more bugs missed by Agamotto: Bug ID 1, 34, 35, 44, 45, and 47 in Table 4.3. Agamotto missed three bugs (ID 1, 45, 47) as they require application-specific custom oracles, and the rest three (ID 34, 35, 44) because symbolic execution could not explore the relevant program paths: *e.g.*, due to recursive functions. The results remain the same after we increase the testing time of Agamotto from one (default) to 12 hours.

For performance bugs, **NVMTest** detected 49 bugs while Agamotto detected 46 bugs, with 22 overlapping bugs. Recall that the performance bug detection depends on program path. The results thus imply that **NVMTest** and Agamotto indeed tested different program paths, showing the differences between symbolic execution and dynamic trace-based approaches. Symbolic execution (Agamotto) can in theory explore all possible paths, but in practice

it does not due to testing budget and hard-to-explore paths (*e.g.*, recursive calls, indirect functions). Dynamic tracing (**NVMTest**) may be lucky to explore some other paths, but do not guarantee the path coverage.

PMTest and XFDetector We also compared **NVMTest** with two annotation-based approaches. Seven programs were tested by **NVMTest**, PMTest, and XFDetector in common: B-Tree, C-Tree, RB-Tree, Hashmap-TX, Hashmap-atomic, Memcached and Redis. For performance bugs, **NVMTest** detects the one bug that PMTest detected. XFDetector does not detect new performance bugs. For correctness bugs, **NVMTest** detects two out of three bugs PMTest/XFDetector found in B-Tree (Bug ID 41) and RB-Tree (ID 42). Both are due to missing logging inside a transaction. In addition, **NVMTest** detected another bug (ID 1) in the PDMK library while testing B-Tree, which was missed by them.

NVMTest missed one bug in `Redis` reported by PMTest/XFDetector. Upon further investigation, it turns out that the bug was 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. **NVMTest** actually detected this store violating an atomicity invariant, and performed the validation. But it does not show any visible divergence. This example particularly shows the benefit of our metamorphic testing based validation, pruning false positives.

Chapter 5

Detecting crash consistency bugs in multi-threaded NVM software

5.1 Proposed Design

Detecting crash consistency bugs in multi-thread NVM software is challenging, since we need to explore at least two dimensions of test space: (1) NVM state space and (2) thread scheduling space. This chapter proposes an NVM-aware thread scheduler for multi-threaded NVM software debugging.

As discussed in section 2.3, our key observation is that multi-threaded NVM bugs only happens when using an volatile value from another thread. To prevent it, lock-based and lock-free fixes can be applied. Both of the fixes guarantee that volatile values must be persisted before another thread using it. Having this in mind, we propose our solution as shown in Figure 5.1.

The input comprises a program and a test case. A test case is a sequential of operations, each of which could be a `insert`, `get` or `delete`. The program is instrumented for tracing, which traces NVM accesses, flush, fence and lock primitives.

The instrumented program and the test case is fed into the single-thread trace search. The output of single-thread trace searching are a prefix of operations and two candidates of

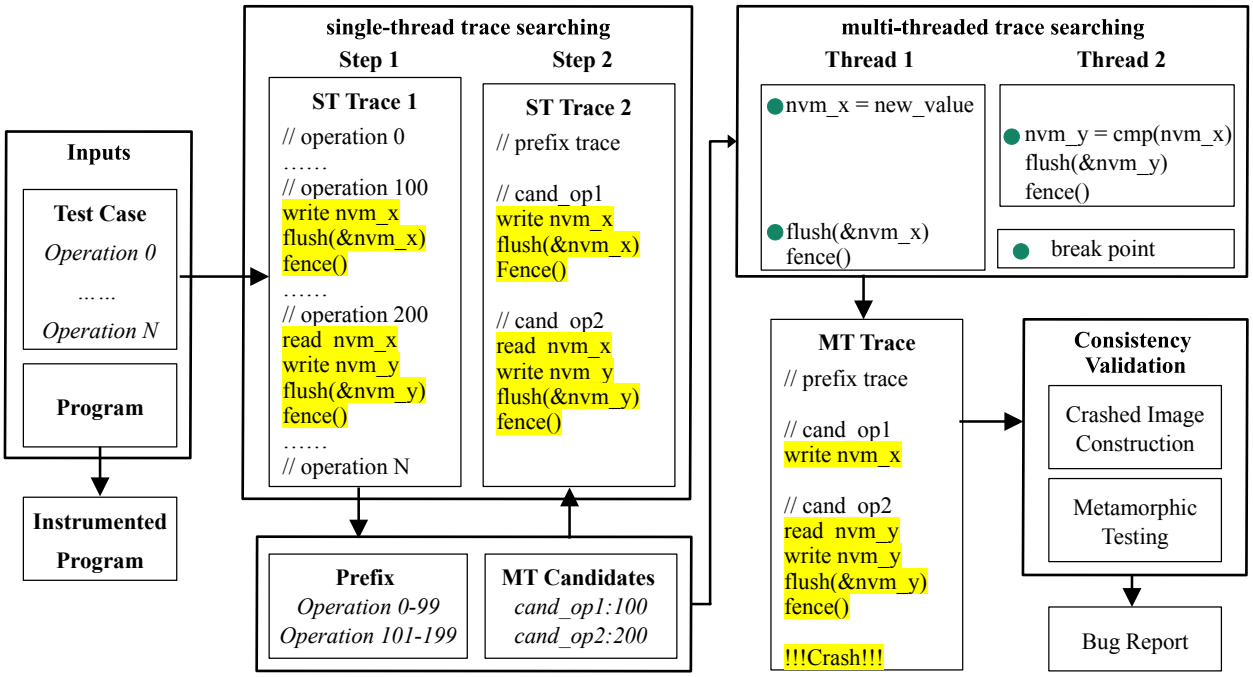


Figure 5.1: Overview of detecting multi-threaded NVM bugs

operations, which will be fed in to the multi-thread searching later. The prefix is a sequence of operations which is to trigger the NVM image to a specific state. The two operation candidates are the two operations which may trigger a multi-threaded NVM bug. The single-thread trace search composes two steps. The first step runs the instrumented program with the test case input in single thread and generates ST Trace 1. The ST Trace 1 consists the trace of all operations in input test case. The the ST Trace 1 is analyzed for generating the prefix and candidates. More specifically, we check whether a pair of operations violates the lock-based or lock-free fix pattern, which is an indicator of a potential multi-threaded NVM bug. If a violation is found, the prefix and the candidates are generated. Using the example in Figure 5.1, the candidates are operation 100 and operation 200, and the prefix is operation 0-99 and operation 101-199. The second step then runs the instrumented program with the prefix and the two candidates in single thread and generate ST Trace 2. The ST Trace only consists the trace of prefix and two candidates. Then we check the trace of the

two candidates to see whether it still violates the lock-based or lock-free fix pattern. If it does, the prefix and candidates will be fed into the multi-thread trace searching.

The output of multi-threaded trace searching is a multi-threaded trace, where a thread uses an unpersisted value and a crash happens before the unpersisted value gets persisted. Using the example in Figure 5.1, thread 1 writes `nvm_x` first, then thread 2 preempts and writes `nvm_y`, which is data-dependend on `nvm_y` and make `nvm_y` persisted. Right after that, a crash happens. The thread scheduling is controlled by setting break points in the corresponding NVM accesses and primitives. After setting the break points, the execution can be deterministically controlled to generate the desired multi-threaded trace.

After getting the multi-threaded trace, a simulated crashed NVM image can be constructed based on it and the metamorphic testing can be applied to check whether there is bug.

5.2 Proposed Evaluation

We plan to evaluate our bug detector using both concurrent persisted indexes and concurrent persistent key-value stores. The persisted indexes include Fast Fair [32], Level Hash [83], CCEH [60], P-ART [48], P-BwTree [48], P-CLHT [48], P-Hot [48] and P-Masstree [48]. The key-value stores include cmap, csmap and robinhood from pmemkv [8]. The evaluation result includes detected multi-threaded bugs, the scalability and bug detection effectiveness.

Chapter 6

Conclusion

The use of NVM technologies is on the rise. The ability to directly access NVMs using regular `load` and `store` instructions provides a new opportunity to build *crash-consistent* software without paying storage stack overhead. However, it is hard to design and implement a correct and efficient crash-consistent program. A NVM bug 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.

Crash consistency testing tools, especially the ones designed to detect correctness bugs in NVM programs, need to inspect a large number of NVM states feasible on a crash — a new dimension of the testing space. Traditional (non-NVM) testing tools explore two dimensional test space of a program input and/or a thread schedule. As the third dimension, an automatic crash consistency testing tool should be able to enumerate all possible NVM states of a program (as a program may crash at any program point) and check if each NVM state is consistent or not.

My dissertation question is: *How to make NVM programming easy?* My research goal is to provide debugging and programming support for NVM software, by designing and implementing an automatic, scalable and precise crash-consistency bug detector for both single-thread and multi-threaded NVM software. The long-term vision of this research it to help our community to build bug-free NVM software. We propose two interrelated research thrusts in this proposal:

Detecting crash consistency bugs in single-thread NVM software To address test space challenge, **NVMTEST** infers a set of *likely program invariants* (hereafter invariants for short) that are believed true to be crash-consistent by analyzing source codes and execution traces. **NVMTEST** then tests only those NVM states that violate an invariant, instead of relying on exhaustive testing or user's manual annotation. To mitigate the test oracle problem, **NVMTEST** takes a novel *metamorphic testing* approach for crash consistency validation. Metamorphic testing [71] fits well in the context of crash consistency of NVM programs as there are strong relations between the outputs of test cases with and without a crash. If a program resuming from an invariant-violating NVM state produces an output different from the executions without a crash, then we can confidently conclude that the program is not crash-consistent, and the invariant violation is indeed a true crash consistency bug.

Detecting crash consistency bugs in multi-threaded NVM software Detecting crash consistency bugs in multi-thread NVM software is challenging, since we need to explore at least two dimensions of test space: (1) NVM state space and (2) thread scheduling space. Our key observation is that multi-threaded NVM bugs only happens when using an volatile value from another thread. To prevent it, lock-based and lock-free fixes can be applied. Both of the fixes guarantee that volatile values must be persisted before another thread using it. Having this in mind, we propose our solution to use single-thread trace searching for NVM racy accesses which violates lock-based or lock-free fix pattern and then validate is in multi-threaded trace searching. After getting the multi-threaded trace, a simulated crashed NVM image can be constructed based on it and the metamorphic testing can be applied to check whether there is bug.

Bibliography

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

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

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

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

[6] NVM Direct. https://github.com/oracle/nvm-direct.

[7] pmem-redis, . https://github.com/pmem/pmem-redis.

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, December 2008.

[17] 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.

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

[19] 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.

[20] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. volume 8, pages 786–797. VLDB Endowment, 2015.

[21] 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.

[22] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 373–386, 2018.

[23] 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. ISBN 9781450368735. doi: 10.1145/3341301.3359637. URL https://doi.org/10.1145/3341301.3359637.

[24] 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.

[25] 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. ISBN 9781450304900. doi: 10.1145/1926385.1926432. URL https://doi.org/10.1145/1926385.1926432.

[26] 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.

[27] 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.

[28] 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.

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

[30] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. Nvram-aware logging in transaction systems. *Proceedings of the VLDB Endowment*, 8(4):389–400, 2014.

[31] 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 *2018 USENIX Annual Technical Conference (USENIX ATC)*, pages 967–979, 2018.

[32] 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.

[33] INTEL. Persistent Memory File System, 2015. URL https://github.com/linux-pmfs/pmfs.

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

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

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

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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. doi: 10.1145/3133907. URL https://doi.org/10.1145/3133907.

[43] 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.

[44] James C. King. Symbolic execution and program testing. In *Proceedings of the Communications of the ACM*, pages 385–394, 1976.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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. ISBN 978-1-931971-218. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes.

[52] 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. ISBN 9781595935915. doi: 10.1145/1294261.1294272. URL https://doi.org/10.1145/1294261.1294272.

[53] 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.

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

[55] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL https://doi-org.proxy.library.stonybrook.edu/10.1145/96267.96279.

[56] 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.

[57] 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.

[58] 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. ISBN 9781595936332. doi: 10.1145/1250734.1250785. URL https://doi.org/10.1145/1250734.1250785.

[59] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software. *Microsoft Research*, 38:39, 2007.

[60] 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.

[61] 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. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/neal.

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

[63] 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.

[64] 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.

[65] 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.

[66] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. *SIGPLAN Not.*, 47(6):521–530, June 2012. ISSN 0362-1340. doi: 10.1145/2345156.2254126. URL https://doi.org/10.1145/2345156.2254126.

[67] 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. doi: 10.1145/3360561. URL https://doi.org/10.1145/3360561.

[68] 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.

[69] 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.

[70] David Schwalb, Tim Berning, Martin Faust†, Markus Dreseler, and Hasso Plattner†. nvm malloc: Memory Allocation for NVRAM. volume 15, pages 61–72, 2015.

[71] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, Sep. 2016. ISSN 1939-3520. doi: 10.1109/TSE.2016.2532875.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] Michal Zalewski. American fuzzy lop, 2014. URL http://lcamtuf.coredump.cx/afl.

[83] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 461–476, 2018.