Finding and Analyzing Crash-Consistency Bugs in Persistent-Memory File Systems

Hayley LeBlanc
University of Texas at Austin

Shankara Pailoor
University of Texas at Austin

Isil Dillig
University of Texas at Austin

James Bornholt
University of Texas at Austin

Vijay Chidambaram
University of Texas at Austin and VMware Research

Abstract

We present a study of crash-consistency bugs in persistent-memory (PM) file systems and analyze their implications for file-system design and testing crash consistency. We develop FLYTRAP, a framework to test PM file systems for crash-consistency bugs. FLYTRAP discovered 18 new bugs across four PM file systems; the bugs have been confirmed by developers and many have been already fixed. The discovered bugs have serious consequences such as breaking the atomicity of rename or making the file system unmountable. We present a detailed study of the bugs we found and discuss some important lessons from these observations. For instance, one of our findings is that many of the bugs are due to logic errors, rather than errors in using flushes or fences; this has important applications for future work on testing PM file systems. Another key finding is that many bugs arise from attempts to improve efficiency by performing metadata updates in-place and that recovery code that deals with rebuilding in-DRAM state is a significant source of bugs. These observations have important implications for designing and testing PM file systems. Our code is available at https://github.com/utsaslab/flytrap.

1 Introduction

Persistent memory (PM) is a new storage-class memory technology that offers extremely low-latency persistent storage and fine-grained access to storage over the memory bus [22,47]. PM technology has long been a focus of research, and more recently has been commercialized by Intel [3]. A number of file systems [8, 14, 17, 18, 24, 25, 28, 43, 45] have been developed that exploit PM's advantages to build faster, safer storage systems.

One of the main responsibilities of a file system is to keep the user's data safe in the event of a crash due to a power loss or a kernel bug. To do so, the file system should be *crash consistent*: it should recover after the failure to a consistent state without losing data the user expected to be persistent [11, 12, 20, 41]. However, there are no open-source tools that can test whether a PM file system is crash consistent without annotating or modifying the file system. Existing crash consistency testing tools like CrashMonkey [38] or Hydra [27] are not compatible with the unique storage stack of PM file systems where the media is directly accessed using processor load and store instructions at fine granularity.

This paper makes two contributions. First, it presents FLYTRAP, a framework for testing the crash consistency of PM file systems (§3). Given a workload, and a target PM file system, FLYTRAP simulates crashes at different points in the workload, creating *crash states* reflecting the on-PM state after the crash; FLYTRAP then mounts the target PM file system on the crash state, and checks if it recovers correctly. FLYTRAP does not require modifying the file system implementation, and is compatible with all PM file systems that implement the POSIX interface. To choose workloads to test, we couple FLYTRAP with a modified version of the ACE [38] workload generator and the Syzkaller [7] gray-box fuzzer. While ACE systematically generates small workloads for FLYTRAP to test, Syzkaller generates random workloads driven by code coverage.

Second, this paper presents an analysis of 18 unique bugs found by FLYTRAP across four PM file systems (§5). We have reported all 18 bugs upstream; all except the two bugs in PMFS (which is not actively maintained) have been confirmed by their developers, and 12 have already been fixed. The bugs have severe consequences such as breaking the atomicity of the `rename()` system call that applications depend on for atomic updates [40]. To the best of our knowledge, this is the largest published corpus of PM file-system crash-consistency bugs; analyzing it yields useful insights for both PM file-system design and efficient crash-consistency testing of PM file systems. For example, we observe that many bugs are logic or design issues in performance optimizations rather than the missing flush/fence bugs that many PM bug-finding tools target [5, 16, 19, 21, 31–33, 39].

Why is testing crash-consistency for PM file systems hard? The unique architecture of PM file systems creates two challenges for effective testing. The first challenge is to intercept writes to the storage media. Existing crash-consistency testing tools such as CrashMonkey [38] intercept writes at the block layer, but PM file systems remove this natural interception point. Instead, they write directly to the media using pro-

cessor store instructions, which would be prohibitively expensive to instrument and require reasoning about the CPU cache hierarchy and store reordering to model faithfully. The second challenge is choosing which crash states to test. CrashMonkey and Hydra [27] only simulate crashes *after* system calls have returned, as most file systems only make strong crash-consistency guarantees after `fsync()` or `sync()` calls. PM file systems take advantage of the low-latency, fine-grained media to offer stronger consistency guarantees by making *every* file-system operation synchronous and durable. This means that to find crash consistency bugs, we must test crashes *in the middle* of system calls. The fine granularity of PM writes (8 bytes) also leads to more possible crash states than traditional file systems with block-sized writes.

FLYTRAP. FLYTRAP tackles these challenges by exploiting insights about the way PM file systems are built. To intercept writes, we observe that the PM file systems we studied all have *centralized persistence functions* that perform writes to PM, rather than using inline assembly at different points in the code. For example, there is a `memcpy()` implementation that uses non-temporal stores to write to PM. FLYTRAP uses Kprobes [4] to intercept these functions and record write IOs without modifying the file system. Intercepting writes at the function level rather than the instruction level greatly reduces overhead. We further reduce this overhead by noting that only data that is explicitly flushed can be used to provide crash-consistency guarantees. Rather than intercepting all writes, FLYTRAP only tracks writes that are explicitly flushed or are written with non-temporal stores.

To choose which crash states to test, we build on this higher-level interception as well as empirical results about the write patterns of PM file systems. Intercepting at the function level allows us to see an entire file-system-level write at once, and so we can coalesce individual 8-byte stores when appropriate; for example, we need not test each 8-byte write of a 1 MB `write()` call separately. We also observe that the set of *in-flight writes* (writes in volatile caches that have not yet been flushed) at any point in a system call is typically small, reducing the number of crash states FLYTRAP must explore. Finally, we observe that the order in which non-overlapping in-flight writes are written to PM does not matter, and so FLYTRAP only considers different subsets of in-flight writes being persisted rather than all permutations.

Generating workloads. Given a workload, FLYTRAP provides a mechanism to generate and test crash states; an orthogonal question is deciding which workloads to explore. CrashMonkey [38] hypothesized that systematically exploring small workloads on small file system states was effective in finding crash-consistency bugs; we sought to test this hypothesis for PM file systems. We modify the Automatic Crash Explorer (ACE) workload generator from CrashMonkey to account for the fact that system calls are synchronous in the tested systems, eliminating the need for `fsync()`. To try to invalidate the hypothesis, we also use the Syzkaller [7] gray-box kernel fuzzer that can generate longer, more complex workloads. We modify Syzkaller to only consider file-system-related system calls and to include recovery code when measuring code coverage.

Testing results and observations. We use FLYTRAP to test four open-source in-kernel PM file systems: PMFS [18], NOVA [45], NOVA-Fortis [46], and WineFS [24]. FLYTRAP finds 18 unique bugs across the four file systems, with two bugs present both in PMFS and WineFS (which builds on PMFS). From these results, we draw some common observations about PM file systems and how to test them:

- The CrashMonkey hypothesis about simple workloads finding many crash-consistency bugs also holds for PM file systems: 15 of 18 bugs are found by the systematic exploration of our modified ACE.
- A majority of the discovered bugs (11/18) could only be found by simulating a crash in the middle of a system call, meaning that existing tools that test only between system calls would not have been effective.
- While a number of recent tools focus on finding missing or duplicate cache flushes and fences in PM applications [5, 16, 19, 21, 31–33, 39], we found that a majority of the bugs (14/18) did not result from such issues, but instead from mistakes in logic (*e.g.,* a metadata item that was not added to a transaction).
- PM file systems increase performance by maintaining some data structures only in DRAM, and rebuilding them when the file system is mounted [24, 25, 45]; FLYTRAP found seven bugs in such code.
- Six bugs arose from developers trying to increase performance by updating metadata in-place, which is much easier to do with the fine-grained access model of PM, rather than inside a transaction.
- NOVA-Fortis [46] contains a number of features not present in NOVA [45] that are intended to increase resilience; interestingly, FLYTRAP found five bugs in these complex features.

The analysis contains a number of other observations, along with a discussion of their implications. To the best of our knowledge, this is the first such analysis of crash-consistency bugs in PM file systems.

In summary, this paper makes the following contributions:

- A set of tools to test crash-consistency of PM file systems, including the FLYTRAP framework (§3)
- A corpus of 18 crash-consistency bugs discovered by these tools across four PM file systems (§4)
- An analysis of discovered crash-consistency bugs, with insights for PM file-system design and crash-consistency testing (§5)

2 Background and Motivation

This section describes file-system crash consistency. It then discusses why crash consistency is important, why testing it for PM file systems is challenging, and why existing tools do not solve this problem.

Crash consistency. A file system is *crash consistent* if it maintains a set of guarantees about its data and metadata after a crash due a power loss or a kernel bug [12, 20, 41]. For example, if there is a crash in the middle of a `rename()` system call, the POSIX standard requires that the file system after recovery should have the file in either the old name or the new name; in other words, rename must be atomic even if there is a crash [6].

Many applications depend on the file system to be crash consistent [40]. Continuing with the rename example, many applications including text editors such as emacs and vim use temporary files to store user data, and rename the temporary files over the original files when the user saves the file. If rename is not atomic, these applications can lose user data in a crash. Unexpected power loss occurs even in professionally-managed data centers [34–37, 42, 44]. Thus, it is important to ensure that file systems are crash consistent.

Persistent memory (PM). Persistent memory technology, recently commercialized as Intel Optane DC Persistent Memory [2,3], combines the properties of traditional storage media and DRAM; it is byte-addressable and connected to the memory bus like DRAM, but provides persistence like traditional storage media. Compared to DRAM, Optane PM provides 0.33× read bandwidth, 0.17× write bandwidth, 2–4× higher read latency, and similar write latency.

In the x86 programming model, PM is accessed via processor load and store instructions. Writes to PM flow through the CPU cache hierarchy like any other memory store, and so do not become immediately persistent. Data can be flushed from CPU caches to persistent media with cache line flush instructions (`clfush`, `clflushopt`, `clwb`), or can bypass cache entirely with non-temporal stores (`movnt`). Because writes to PM are processor stores, they are also subject to CPU store reordering, and so must be surrounded by store fences when preserving order is important for consistency. We say that data whose cache line has been written back, or which was written using non-temporal stores, is *flushed* to PM once a subsequent store fence instruction has executed, as it is guaranteed to reach media before any future writes. We term a write that has not yet been flushed to persistent media an *in-flight* write; in-flight writes may be lost in the case of a crash. If there are multiple in-flight writes, they may be written to persistent media in any order.

PM file systems. PM file systems differ from traditional file systems in a few important aspects. First, traditional file systems have a long software path for writes along the storage stack; all writes go through the block layer, and most through the page cache, before hitting the storage media. In the case of PM, writes are performed using processor stores, providing a significantly shorter path from file system to storage media. Second, while traditional file systems buffer updates in memory before writing them to storage, PM file systems tend to synchronously write to persistent media given the low latency and high bandwidth of PM [18, 24, 25, 28]. As a result, every system call leads to persistent writes, not just `fsync()` and `sync()`.

These design decisions lead to two challenges when testing PM file systems for crash consistency:

- *Intercepting writes is challenging*. Existing tools to test crash-consistency intercept writes at the block layer and use the intercepted write I/O to construct crash states. PM file systems remove this natural interception point. PM file systems write to the persistent media using processor store instructions; the small granularity of the store instruction (8 bytes) also increases the number of instructions that would need to be intercepted. A naive solution such as intercepting each store instruction (even if we could correctly identify stores to PM) would be prohibitively expensive.

- *Increase in crash states to explore*. While existing crash-consistency testing frameworks only simulate crashes at system-call boundaries, testing PM file systems requires simulating crashes *in the middle* of system calls. This is because PM file systems persist state synchronously in file-system operations, instead of buffering them until `fsync()`. The small granularity of writes also increases the number of potential crash states resulting from each write or metadata operation. Testing crash consistency in PM file systems requires simulating crashes in the middle of each file-system operation, versus just after `fsync()` or `sync()` calls.

Why current tools are not enough. CrashMonkey [38] and Hydra [27] can be used to check the crash consistency of traditional file systems. However, they only simulate crashes at the end of system calls such as `fsync()` because that is when persistence is guaranteed. However, PM file systems have stronger persistence guarantees, and we need to simulate crashes in the *middle* of the system call, not only at the end. It is not straightforward to do this since the persistence guarantees are unclear when crashing in the middle of a system call. Moreover, CrashMonkey and Hydra cannot intercept PM writes.

Yat [29] is an internal tool used by Intel to test the crash consistency of PM software, including PM file systems. Yat uses a hypervisor-based recording technique to log all writes to PM and reconstructs crash states to check. It produces an extremely large number of crash states; one of the three reported test workloads used with PMFS would take over five

years to run fully. Yat is not open source, and the exact set of bugs it found is not available.

A number of tools are available for checking PM software for *PM programming errors*: errors associated with persisting data to PM, such as missing or unnecessary cache-line flushes or store fences [5, 16, 19, 21, 31–33, 39]. PMTest [33], XFDetector [32], Pmemcheck [5], and PMDebugger [16] require manual annotation, which we seek to avoid. All of them are primarily focused on low-level PM programming errors or narrow classes of logic bugs that are directly related to PM management. For example, Witcher [19] looks for fine-grained "persistence atomicity violations" in which sequences of writes to PM that are assumed to be atomic can be interrupted by a crash. Agamotto [39], which uses symbolic execution to track the state of PM, also allows developers to provide custom bug oracles to check for precise, low-level properties about a program; for example, that a specific type of structure is always modified within a transaction. We are interested in checking the higher-level crash consistency guarantees provided by PM file systems and the POSIX interface without requiring specifications from developers, so these tools are not sufficient.

In summary, there is a strong need for a tool that can be used to efficiently test PM file systems for crash consistency without requiring manual annotations.

3 FLYTRAP

We present FLYTRAP, a new framework to find crash-consistency bugs in PM file systems. FLYTRAP tackles the challenges outlined in §2 by exploiting two characteristics of PM file systems: centralized persistence functions and limited number of in-flight writes. FLYTRAP can be run on all PM file systems implementing the POSIX API, and it neither requires manual annotations nor modifying the file-system code.

FLYTRAP takes as input a workload to execute against a chosen file system and outputs bug reports with enough details to reproduce the bug. We have developed two workload generators: one based on the ACE tool [38] for exhaustively enumerating file-system operations, and one based on the Syzkaller gray-box fuzzer [7]. Together, these tools offer an automated and off-the-shelf approach for discovering bugs in PM file systems.

This section first provides an overview of the end-to-end bug finding process (§3.1). It then describes how FLYTRAP tackles the main challenges in testing PM crash consistency (§3.2). It then presents the architecture of FLYTRAP (§3.3) and workload generators (§3.4), followed by a discussion of the limitations of the framework (§3.5).

3.1 Overview

Figure 1 provides an overview of the FLYTRAP framework. FLYTRAP is a record and replay framework. It first runs a given workload (a sequence of file-system operations) and records the writes made by the file system. It then replays these writes to create crash images, which represent the state of the system if it had crashed at different points during the workload. FLYTRAP constructs crash images for crash points both during and after system calls. FLYTRAP mounts the target file system on the crash image, lets it recover, and then checks whether it has recovered to a consistent state. Consistency is workload-specific; for example, if the crash happened in the middle of a rename, FLYTRAP will check whether the old file or the new file exists. Both files missing or both files existing would result in a bug report. The bug report contains the workload, the crash point, the file-system state after crash, what was expected, and the file system and kernel version.

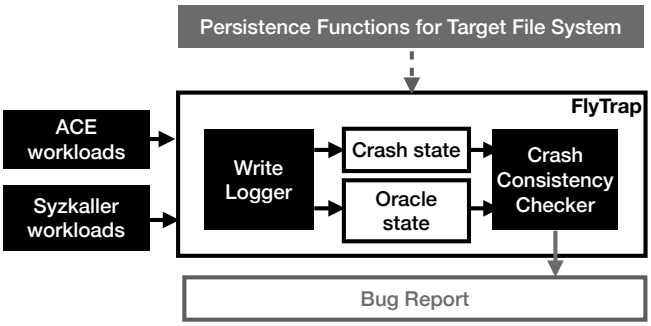

Figure 1: **Architecture**. Given a target file system and its persistence functions, FLYTRAP uses workloads from both ACE and Syzkaller to test the file system. FLYTRAP produces bug reports with enough detail to reproduce the bug.

The ACE workload generator is based on a hypothesis outlined in the CrashMonkey work [38]: that testing small workloads (with a few file-system operations) on a newly created file system is useful for finding crash-consistency bugs. We use the ACE workload generator with FLYTRAP to test if this hypothesis holds for PM file systems. We modify ACE to take into account the properties of PM file systems, such as all operations being synchronous.

To try to disprove the hypothesis, we also use the Syzkaller gray-box fuzzer to generate long, complex, and randomized workloads for FLYTRAP. Syzkaller tries to generate random tests that increase code coverage. We modified Syzkaller to take the recovery code into account when calculating code coverage.

Finally, we cluster bug reports based on their text, such as the error (*e.g.,* missing file) and the workload. This allows us to efficiently find and report unique bugs to developers.

3.2 Challenges

FLYTRAP tackles the challenges in testing PM file systems for crash consistency (§2) by exploiting two characteristics of PM file systems.

Intercepting writes. Intercepting the write traffic of a PM file system is more complex than a traditional file system because PM file systems do not use the usual block layer to write data to the storage media, removing a convenient interception point. Instead, PM file systems use processor store, fence, and cache flush instructions directly. However, all in-kernel PM file system implementations we examined include *centralized persistence functions* to interact with the PM device rather than directly using inline assembly at each write location. In particular, every file system we evaluate in this paper offered four persistence functions: non-temporal `memcpy`, non-temporal `memset`, cache line flush, and store fence. These abstractions simplify reasoning about PM semantics and potentially enable portability to other architectures.

We exploit this observation to simplify intercepting writes by requiring the file system developer to provide FLYTRAP with annotations to identifies these four persistence functions for their file system. FLYTRAP then automatically instruments these functions at run time using the Kprobes [4] debugging mechanism in the Linux kernel.

Increased number of crash states. PM file systems make very fine-grained writes to storage media. Also, since these systems perform operations synchronously, they provide strong guarantees about crash consistency at all times, not just after `fsync()`. As a result, a workload can result in significantly more crash states that are interesting to test in a PM file system than in a block-based file system. We handle the large number of crash states by exploiting three observations:

- *Data required for crash consistency must be flushed*. While the PM file system may write a lot of data, only data that is explicitly flushed or is written via non-temporal stores can be relied upon to be persistent. CrashMonkey [38] and Hydra [27] leverage a similar observation to only simulate crashes after `fsync()` and `sync()`. FLYTRAP exploits this observation to only track writes when they are flushed — it collects write information only for cache flushes and non-temporal writes. This drastically reduces the overhead of logging and the number of functions FLYTRAP must track.

- *Order of in-flight writes does not matter*. If there are two in-flight writes *A* and *B* (data in volatile CPU caches that has not been flushed yet), the order in which A and B are persisted does not matter. A crash state where *A* is written first followed by *B* is equivalent to a crash state where *B* is followed by *A*. If *A* and *B* are writes to the same cache line, x86 enforces sequential ordering. As a result, we do not have to consider re-orderings of in-flight writes. FLYTRAP creates crash states using different subsets of in-flight writes, taking into account writes to the same cache line.

- *Small number of in-flight writes*. The number of generated crash states depends on the number of in-flight writes. In our testing, we observed that the number of in-flight writes at any given point of time is small: across the two workload generators (Section 3.4) and four file systems, the average number of the in-flight writes is three, and the maximum is ten. The only exception was the `write()` system call in PMFS, which we observe to have up to 20 in-flight writes when writing a large amount of data. This allows FLYTRAP to exhaustively enumerate and check all subsets of in-flight writes.

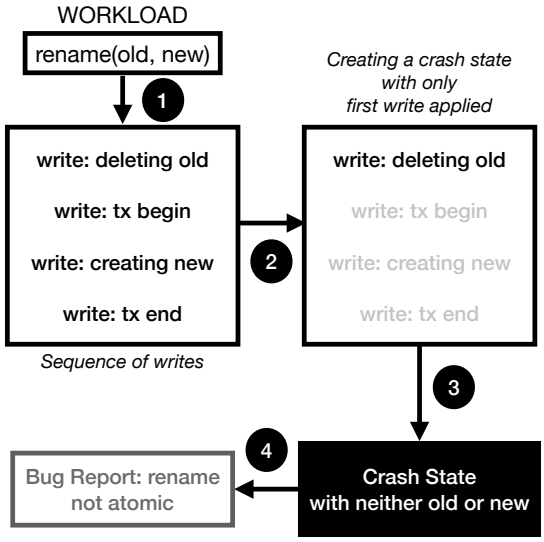

Figure 2: **FLYTRAP workflow**. The figure shows how crash consistency is tested using a simple `rename()` workload. In this example, the old file being deleted is updated in-place, while the new file creation happens inside a transaction. 1) FLYTRAP runs the workload on the target file system, and logs a sequence of PM writes, flushes, and fences. For the sake of simplicity, assume that all writes are flushed and there is a store fence at the end of the system call. 2) FLYTRAP creates a crash state where only the old file is deleted; the other writes are lost in the crash. 3) The consistency checker finds that both the old file and the new file are missing 4) FLYTRAP creates a bug report. This bug in NOVA discovered by FLYTRAP (bug 4).

Taken together, these observations allow FLYTRAP to track only flushed writes (leading to low overhead), consider only subsets (and not permutations), and exhaustively test all the subsets resulting from in-flight writes.

3.3 FLYTRAP Architecture

FLYTRAP is built on top of the CrashMonkey framework [38], which consists of two kernel modules and user-space utilities to facilitate checking crash states. We adapt CrashMonkey's user-space utilities to target PM file systems; the two kernel modules are specific to block devices and are not compatible

with PM. We replace them with modules based on Kprobes, a Linux kernel debugging utility, to record writes to PM.

Given a workload and a target file system, FLYTRAP proceeds in three steps (Figure 2): (1) run the workload and log the writes made by the file system; (2) construct crash states; (3) check each crash state and generate a bug report if required. We now describe each step in more detail.

Logging writes. Recall from Section 3.2 that FLYTRAP requires PM file system developers to identify the centralized persistence functions. Given the locations of these functions, FLYTRAP uses Kprobes [4] to automatically instrument them at run time. Each time one of these functions is invoked by the file system, the instrumentation logs the operation and its arguments (including the current value of the cache line for cache line flushes). The user-space test harness also inserts markers into this log to record the start and end of each system call. This approach requires no code changes to the file-system implementation other than to prevent the compiler from inlining the persistence functions. In our experience, identifying these functions was simple, and we expect it to be even simpler for file-system developers who likely already know where the persistence functions are in their implementation.

Constructing crash states. Given a workload, FLYTRAP can simulate crashes both after and during system calls. FLYTRAP replays a workload by walking through the log of writes. Whenever it encounters a cache line write back or non-temporal store, it adds it to an in-flight vector. When it encounters a store fence, it flushes the contents of the in-flight vector. To generate crash states after system calls, FLYTRAP replays all writes that have been flushed by the end of the system call. To generate crash states during system calls, it simulates a crash immediately before each store fence and creates a set of crash states by replaying each subset of the in-flight vector. The number of crash states is thus dependent on the number of in-flight writes; if there are n in-flight writes, there will be $2^n - 1$ crash states. As noted in Section 3.2, we have observed that n is small in practice, allowing FLYTRAP to apply this exhaustive testing strategy. Since a small number of in-flight writes is not a guarantee and we occasionally see larger sets while using Syzkaller, FLYTRAP can place a configurable cap on the number of writes to replay. We find that in practice, even a cap of two writes is sufficient to reveal many bugs (§5.1).

Testing each crash state. To check file-system consistency, FLYTRAP first mounts the target file system on each crash state, which is itself a useful consistency check as failure to mount is a serious bug. Once successfully mounted, the file-system state is compared against two oracle file-system states: a *before* state representing the file-system state before the crashing system call began and an *after* state representing the file-system state if the system call during the crash successfully completed. After recovery, the file-system state should match either the before or after state. This check validates properties implied by POSIX or widely expected by users in practice [11, 40]. Finally, to ensure the file system is in a usable state, FLYTRAP creates files in all directories and then deletes all files. If any of these checks fail, the checker reports a bug describing the inconsistency and the corresponding crash state.

To make this implementation efficient, we construct oracle file systems incrementally as the log is replayed because many crash states share the same oracle state. We also construct crash states in-place on the PM device, mutating them to generate each crash state rather than recreating them from scratch each time. Because the consistency checks mutate the state, we reuse our logging infrastructure to record an undo log for these mutations and roll back the changes when advancing to the next crash state.

3.4 Workload Generation

Given a workload, FLYTRAP generates crash states and tests them for consistency. An orthogonal challenge is generating workloads for FLYTRAP to test. The CrashMonkey work [38] introduced the hypothesis that small workloads on new file systems are useful in finding crash-consistency bugs. While this hypothesis was true on traditional block-based file systems, we aim to test whether it holds on PM file systems. To this end, we modify CrashMonkey's Automated Crash Explorer (ACE), which systematically explores workloads of a given size, to work with FLYTRAP. To disprove the hypothesis, we modify the Syzkaller [7] gray-box fuzzer to work with FLYTRAP. Syzkaller generates long, complex, randomized workloads while aiming to improve code coverage.

3.4.1 Automatic Crash Explorer

We used a modified version of ACE [38] to systematically generate workloads for FLYTRAP. ACE was designed to exhaustively generate workloads of a certain structure to test traditional file systems. It focuses on short workloads with frequent `fsync()`, `fdatasync()`, or `sync()` calls, as these are required to obtain strong crash-consistency guarantees in traditional file systems. Given a sequence length n, ACE generates workloads with n core file-system operations over a small, predetermined set of files, then fleshes them out by satisfying dependencies and adding `fsync()`, `fdatasync()`, or `sync()` operations. A workload with n core system calls is called a "seq-n" workload.

We modify ACE in the following manner for FLYTRAP. First, since system calls in the `fsync` family do not have the same crash-consistency significance in PM file systems as they do in traditional file systems, we remove them from our version of ACE. This significantly reduces the number of workloads generated because we do not need to consider workloads that differ only by the sync calls they use. We also remove system calls and options that are not supported by the

file systems under test. For example, NOVA only supports the `FALLOC_FL_KEEP_SIZE` flag for the `falloc` system call, and it does not support extended attributes. Removing these unsupported options further reduces the number of workloads that ACE generates.

We test all seq-1 and seq-2 workloads, as well as the subset of seq-3 workloads containing only `pwrite()`, `link()`, `unlink()`, and `rename()` calls (i.e. the "seq-3 metadata" workloads in the CrashMonkey work [38]) to make testing tractable. Our modified version of ACE generates 56 seq-1 tests, 3136 seq-2 tests, and 50650 seq-3 metadata tests.

3.4.2 Syzkaller

We modify Syzkaller [7], a state-of-the-art gray-box kernel fuzzer, to generate workloads for FLYTRAP. As is standard in gray-box fuzzing, our fuzzer starts with an initial set of test cases (seeds) and uses genetic programming to generate new tests for FLYTRAP from those seeds. FLYTRAP tests each generated workload and reports back whether the workload produced a crash-consistency bug along with the code coverage achieved on the target kernel. If the workload covered new parts of the kernel, the fuzzer adds it to its set of seeds and generates new workloads from it.

Syzkaller generates workloads by randomly selecting sequences of core file-system operations and their argument values. It generates syntactically and semantically valid workloads by using a detailed template for each system call that specifies more precise *qualified type* information [13] for the call's arguments. For example, the template for `open` specifies that it returns a file descriptor, not just an arbitrary integer; the template for `write` specifies that its first argument is also a file descriptor rather than an arbitrary integer (which would be very unlikely to be a valid file descriptor).

To adapt Syzkaller to our setting, we restrict it to only generate workloads that contain core file-system operations, and replace its workload executor with a custom one. Our executor invokes FLYTRAP on each workload and records code coverage both before the crash and during recovery. We add a workload to the seed set if it achieves new code coverage on either side of the crash.

Like many fuzzers, Syzkaller can quickly generate many bug reports that are duplicates—we found that Syzkaller would frequently generate different workloads that triggered the same bug and produced similar bug reports. In our setting, this duplication also arises when multiple crash states for the same workload trigger the same bug, producing similar bug reports. To address this problem, we extended Syzkaller to automatically triage bug reports generated by FLYTRAP during fuzzing. We use a simple triaging procedure that clusters bug reports by lexical similarity. Whenever FLYTRAP generates a bug report, we compute the report's word vector v (based on its text), and then compute the distance from v to the closest cluster C of previous reports. If the distance from v to C is below a predefined threshold, we add the report to C, otherwise we create a new cluster. This simple heuristic was more effective than more complex ones we tested. We also updated Syzkaller to display these bug report clusters (along with the workload that triggered each bug report) in its UI dashboard to make them easier for users to debug.

3.5 Limitations

FLYTRAP has several limitations. First, it can miss bugs because it does not explore all workloads nor all crash states resulting from a given workload. FLYTRAP only tests some subsets of the data in `write()` system calls and considers cache line flushes atomic with respect to crashes. FLYTRAP only tracks flushed writes; a PM write that is never flushed is invisible to FLYTRAP, and it could result in FLYTRAP missing a bug. For example, consider file-system writes to A, B, C, and a bug where the file system never flushed A. FLYTRAP would not explore a crash state where A and B are persisted, but C is not, and so if this state created inconsistency FLYTRAP would not detect it. Second, FLYTRAP does not support checking concurrent workloads. It assumes that the operations performed by the file system in each workload are deterministic. In our testing so far, we have not observed any non-determinism in the file systems that impacted our ability to find and reproduce bugs with FLYTRAP. Third, FLYTRAP assumes that the PM file system has centralized persistence functions. A PM file system that uses in-line assembly to update PM at various locations in its code would not be compatible with FLYTRAP.

Despite these limitations, we believe that FLYTRAP is a useful addition to the set of tools for building robust PM file systems. In particular, the level of automation provided by FLYTRAP allows developers to test new or in-development PM file systems efficiently.

4 Testing PM File Systems

In this section, we evaluate FLYTRAP's effectiveness at finding bugs across different PM file systems. We describe our experimental setup (§4.1) and present an evaluation of FLYTRAP along with a comparison of ACE and Syzkaller as workload generation strategies (§4.2). We provide a brief overview of the bugs found by FLYTRAP (§4.3).

4.1 Experimental setup

File systems. We considered seven open-source PM file systems for testing with FLYTRAP: NOVA [45], NOVA-Fortis [46], PMFS [18], WineFS [24], Strata [28], Assise [8], and ext4-DAX [1]. ext4-DAX is a modification of the ext4 file system for PM. It shares most of its code with ext4 and has the same crash consistency guarantees, so tools like CrashMonkey or Hydra that are designed for block-based file systems

are a better fit for testing ext4-DAX. Strata and Assise are kernel-bypass file systems implemented in user space. Neither Strata nor Assise currently support recovering from arbitrary crashes. As a result, we chose to focus on the remaining four systems: NOVA, NOVA-Fortis, PMFS, and WineFS.

Test infrastructure. All experiments described in this paper were run on QEMU/KVM virtual machines running Debian Stretch. Each VM is allocated one CPU (except for those testing WineFS, which requires four CPUs) and 8 GB of RAM. Each VM also has two 128 MB emulated PM devices, which are used to execute the workload, construct the oracle file system, and check crash states.

We run ACE-generated workloads on a single Amazon EC2 m5d.metal instance with 96 vCPUs, 384 GB memory, and four 900 GB NVMe SSDs. We use these resources to check multiple file systems using workloads of multiple sequence lengths in parallel. We run seq-1 and seq-2 tests on individual VMs; we split the seq-3 metadata workloads across 10 VMs and ran them in parallel. All file systems were checked using seq-1 and seq-2 workloads, and all but WineFS were run on seq-3 metadata tests. At the time we ran these experiments, WineFS had a bug that prevented creation of the number of files some seq-3 tests require. The number of in-flight writes at any time during ACE tests is consistently low, so we do not place a cap on the number of crash states for ACE.

To evaluate FLYTRAP with Syzkaller, we run four Chameleon Cloud [26] bare metal instances, which have two Intel Xeon Gold 6240R CPUs each with 24 cores and 48 threads, as well as 192 GB RAM, and 480 GB storage. Each host fuzzed a different file system using 15 virtual machines. Each fuzzer starts with an empty set of seeds. Syzkaller-generated tests can be long and generate many crash states, so to avoid the fuzzer getting stuck, we run FLYTRAP with a cap of two writes per crash state; as §5.1.2 observes, this cap does not affect its ability to find bugs in practice.

4.2 Evaluation

ACE tests. For each file system under test, FLYTRAP took about 10–15 minutes to run the seq-1 workloads, 7–11 hours to run the seq-2 workloads, 16–26 hours to run the seq-3 metadata workloads in parallel (160–260 total CPU hours). The number of crash states to check on each workload varies as much as 3× between file systems, with PMFS generally checking the most and WineFS checking the fewest. Overall, FLYTRAP found 15 bugs using ACE tests across all four file systems.

Syzkaller. We ran FLYTRAP with Syzkaller for 18 hours on 15 VMs, for a total of 270 CPU hours spent fuzzing. During this time, FLYTRAP checked over 40 million crash states across all four system calls, finding 18 unique bugs. Three of these bugs cannot be found with ACE-generated workloads.

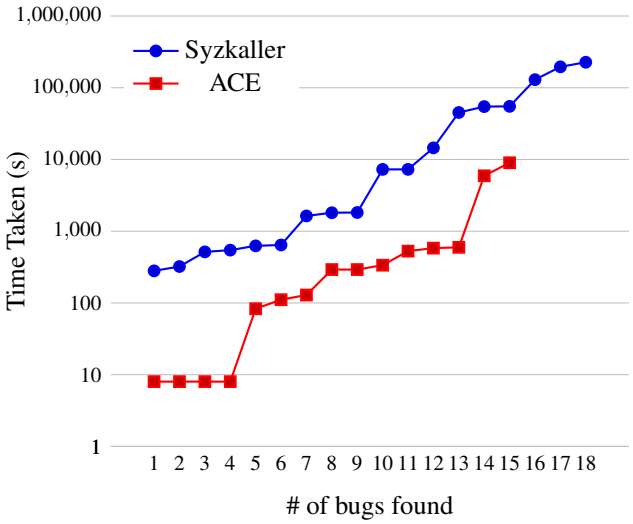

Figure 3: Cumulative time taken to find crash-consistency bugs by ACE and Syzkaller.

Comparison. We ran Syzkaller and ACE on each file system and recorded the cumulative CPU time taken to find all bugs when using each workload generator. Figure 3 shows the result of this experiment. ACE finds the first 15 out of 18 bugs in less than three CPU hours total, but is unable to find the final three bugs. Syzkaller, on the other hand, takes almost 20× more CPU time than ACE to find the first 10 bugs and almost 6× more CPU time to find all the bugs the ACE tester finds. However, when we let Syzkaller run for an additional 27 CPU hours, it is able to find three additional bugs that are not detected by ACE. ACE misses these three bugs because they do not conform to the patterns that it uses to generate workloads. For example, two of these bugs create two open file descriptors to the same file and modify the file's contents through both file descriptors, but ACE does not generate workloads with multiple file descriptors for the same file.

While the results of this experiment indicate that Syzkaller has greater overall bug finding capability than ACE, the ACE tests are considerably more resource efficient. This suggests that the ACE tests can be run locally and iteratively to find bugs during file-system development, whereas Syzkaller should be run for a long time in an environment with ample compute resources (such as the cloud) for more comprehensive crash-consistency testing.

4.3 Results

Crash-consistency bugs. Using ACE- and Syzkaller-generated tests, FLYTRAP finds 18 unique crash-consistency bugs across all four tested file systems. The number of unique bugs is based on the number of separate fixes required to patch

System call	# of bugs
mkdir	1
creat	1
rmdir	1
fallocate	1
unlink	2
link	3
rename	4
truncate	5
write/pwrite	5

Table 1: The number of bugs associated with each system call. Some bugs impact all file system operations, and some impact multiple system calls.

all of the bugs, not different user-visible consequences. Two bugs are found in both WineFS and PMFS (since WineFS is built off PMFS), for a total of 20 bugs.

Table 2 describes the consequences of each bug and the system calls they affect. The bugs are classified as either logic or PM errors (described further in §5.1). FLYTRAP found eight bugs in NOVA, four bugs in NOVA-Fortis, two bugs in PMFS, two bugs in WineFS, and two bugs in both PMFS and WineFS (as WineFS is an extension of PMFS). Many of these bugs have serious consequences that violate the crash consistency guarantees of the file systems. Three of the bugs prevent the file system from being mounted entirely. Two impact the atomicity of `rename()`, which many applications rely on [40]. Many others cause data loss or prevent a user from accessing files entirely.

The bugs impact a wide variety of system calls, as Table 1 shows. Many bugs are located in common metadata-handling or recovery code and thus impact multiple system calls. `rename()`, `write()`/`pwrite()`, `link()`, and `truncate()` are especially bug-prone and are affected by multiple bugs.

Non-crash consistency bugs. While working with FLYTRAP, we also found an additional eight non-crash-consistency bugs not included in Table 2. These bugs occur during regular execution and do not require a crash to be exposed. We were able to find these bugs because they caused KASAN errors, segmentation faults, or incorrect behavior that our consistency checks could detect. For example, using the fuzzer, we discovered that NOVA does not properly handle `write()` calls where the number of bytes to write is extremely large; it will allocate all remaining space for the file, causing most subsequent operations to fail. FLYTRAP detected this bug because it found that new files could not be created in crash states following this write.

5 Bug Analysis

This sections presents an analysis of the 18 crash-consistency bugs found by FLYTRAP (Table 2). To the best of our knowledge, this is the first such corpus of crash-consistency bugs in PM file systems. We make some observations about common patterns we found among the bugs (§5.1), and distill lessons for building and testing PM file systems (§5.2).

5.1 Observations

We first present observations about the nature of the crash-consistency bugs found by FLYTRAP, and then present observations about crash-consistency testing.

5.1.1 Nature of crash-consistency bugs

Observation 1: A majority of the observed crash-consistency bugs are logic issues rather than PM programming errors. When we began working on FLYTRAP, we expected that most bugs would be due to the subtleties of the persistent-memory programming model that existing PM application research has focused on: interaction with the cache hierarchy, CPU store (re)ordering, etc. However, the majority of bugs we found—14 of 18—are actually due to logic bugs in manipulating file-system data structures rather than mistakes in managing persistent memory. The "type" column in Table 2 classifies bugs into logic bugs or PM bugs. Logic bugs are issues that cannot be fixed by adding cache line flushes or store fences. These results suggest that it is not sufficient for a file-system crash-consistency testing tool to focus on exploring the persistence behavior of individual writes and reorderings; it must also exercise the file-system data structures and recovery mechanisms, and check higher-level consistency properties that cannot be validated at the level of individual writes.

Observation 2: In-place update optimizations are a common source of crash consistency bugs. One of the allures of persistent memory is that programs can access it as memory, performing fine-grained reads and writes directly rather than coalescing them into larger block-sized IO operations on slower storage media. This design makes it possible in principle to reduce the overheads of traditional consistency mechanisms like journaling by manipulating on-disk data structures directly. All four file systems we tested use a journal for consistency, but all have performance optimizations to bypass the journal and perform in-place updates in certain circumstances. For example, NOVA updates the link count of a file by updating a per-inode log. Appending to this log is done via a journalled transaction, but if the previous operation on the file also updated its link count, NOVA may modify that log entry in place instead.

We found these optimizations to be particularly error-prone: 6 of 18 bugs in Table 2 are caused by in-place updates. For

Bug #	File System	Consequence	Affected system calls	Type
1	NOVA	File system unmountable	All	Logic
2	NOVA	File is unreadable and undeletable	`mkdir, creat`	PM
3	NOVA	File system unmountable	`write, pwrite, link, unlink, rename`	Logic
4	NOVA	Rename atomicity broken (file disappears)	`rename`	Logic
5	NOVA	Rename atomicity broken (old file still present)	`rename`	Logic
6	NOVA	Link count incremented before new file appears	`link`	Logic
7	NOVA	File data lost	`truncate`	Logic
8	NOVA	File data lost	`fallocate`	Logic
9	NOVA-Fortis	Unreadable directory or file data loss	`unlink, rmdir, truncate`	PM
10	NOVA-Fortis	File is undeletable	`write, pwrite, link, rename`	Logic
11	NOVA-Fortis	FS attempts to deallocate free blocks	`truncate`	Logic
12	NOVA-Fortis	File is unreadable	`truncate`	Logic
13	PMFS	File system unmountable	`truncate`	Logic
14 &15	PMFS&WineFS	Write is not synchronous	`write, pwrite`	PM
16	PMFS	Out-of-bounds memory access	All	Logic
17 &18	PMFS&WineFS	File data lost	`write, pwrite`	PM
19	WineFS	File is unreadable and undeletable	All	Logic
20	WineFS	Data write is not atomic in strict mode	`write, pwrite`	Logic

Table 2: Bugs found by FLYTRAP, their consequences, and the system calls that they affect.

example, in bug 4, NOVA's `rename` implementation removes the directory entry from the parent inode in-place but journals the other metadata changes, allowing the file to be lost in a crash before the journal transaction commits.

Fixing these bugs often requires journalling more data, which is not free. To quantify the impact of fixing such bugs, we compared the performance of NOVA before and after fixing two rename atomicity bugs (4 and 5) by journalling more metadata. We tested both versions on Intel Optane DC Persistent Memory media. In a microbenchmark that repeatedly creates and renames a file over the top of an existing file, the fixed version of NOVA is 25% slower. A more real-world metadata-intensive benchmark (checking out different stable versions in the Linux kernel git repository) shows negligible overhead (<1%). In some cases, journalling can even be better than in-place updates. The fix for bug 6 replaces an in-place update in `link` with extra logging, but makes a microbenchmark that repeatedly creates links to a file 7% faster, likely because checking whether the in-place update is safe requires an extra read from the media.

Observation 3: Rebuilding volatile state during crash recovery is error-prone. In a traditional file system, crash recovery scans on-disk structures like journals and updates the durable state to match. PM file systems often perform more work during recovery. As a performance and write-endurance optimization, they avoid durably storing some metadata (e.g., free page lists or file sizes) and instead rebuild it in volatile memory at mount time. This rebuilding code is subtle because it must account for potential inconsistencies or partial states after a crash, and we found that 8 of the 18 bugs in Table 2 were in such code. For example, bug 13 is caused by a crash during a `truncate` system call on PMFS. The implementation first stores information about the operation in a "truncate list"; if the system crashes before the truncation is complete, the truncate list can be replayed to finish the operation. However, replaying truncations requires accessing the free page list, which PMFS rebuilds in volatile memory during recovery, but only *after* replaying the truncate list. Attempts to replay truncations therefore cause a null pointer dereference when accessing the free page list.

Rebuilding volatile state is more complex in PM file systems that maintain *per-CPU* volatile state to improve scalability. In bug 19, WineFS failed to properly index into an array of per-CPU journals that were read during crash recovery. FLYTRAP can even find bugs in rebuilding code in the absence of crashes, because the crash states it tests include ones where *all* writes are durable. For example, bug 7 involves modifying the same file through multiple file descriptors on NOVA, which creates log entries out of chronological order, violating assumptions in the rebuilding code and causing data loss during `truncate` operations.

Observation 4: Resilience mechanisms to recover from

Observation	Associated bugs
Many bugs occur due to logic or design issues, not PM programming errors.	1, 3–8, 10–13, 16, 19, 20
The complexity of performing in-place updates leads to bugs.	4–7, 14, 15
Recovery code that deals with rebuilding in-DRAM state is a significant source of bugs.	1, 3, 7, 11, 13, 16, 19
Complex new features for increasing resilience can introduce new crash consistency bugs.	2, 9–12
A majority of observed bugs can only be exposed by simulating crashes during system calls.	3–6, 9–13, 19, 20
Short workloads were sufficient to expose many crash consistency bugs.	1–6, 9–20
Many bugs were exposed by replaying a few small writes onto previously persistent state.	3–6, 9–13, 19, 20

Table 3: Observations and the bugs associated with them.

media failures can introduce new crash-consistency bugs. NOVA-Fortis [46] is an extension of NOVA that adds fault detection and tolerance for media errors and software bugs by (among other techniques) replicating and checksumming inodes and logs and checksumming file data. While NOVA-Fortis is not explicitly designed to increase crash resilience, we tested these features to see if it is more tolerant of crashes than the original NOVA.

We found that NOVA-Fortis has all the same crash-consistency bugs we found in the original version of NOVA, and in addition has five new bugs caused by the added complexity of maintaining redundant state and checksums. A common theme in these bugs is that data and metadata modifications are often not atomic with checksum and replica updates, allowing checksum validation to fail (and render a file inaccessible) even if the file system is consistent and data intact.

5.1.2 Crash-consistency testing in PM file systems

Observation 5: A majority of observed bugs require simulating crashes during system calls. Current crash-consistency testing tools for traditional file systems, like CrashMonkey [38] and Hydra [27], insert crashes only at the end of persistence system calls (`fsync`, `fdatasync`, etc.). This heuristic exploits the fact that most POSIX APIs only make crash-consistency guarantees after persistence operations, so intermediate states are unlikely to violate the specification. It allows these tools to scale to test larger workloads, and does not appear to cause them to miss bugs: CrashMonkey has a mode to insert crashes during system calls, but it did not find any additional bugs in the file systems tested in that work.

We found that this same heuristic does not work for PM file systems. 11 of the 18 bugs in Table 2 require a crash to occur during a system call. This is a corollary of our observation that most PM file systems intend to implement most system calls synchronously, and so their effects are fully persistent by the end of the system call. For example, the rename atomicity bugs in NOVA (bugs 4 and 5) arise when a crash during the system call leaves only some writes persisted. Waiting until a persistence point to check consistency would not discover these bugs, as NOVA correctly flushes all writes by the end of the rename operation.

Observation 6: Short workloads suffice to expose many crash-consistency bugs. We use ACE [38] to systematically generate test workloads. ACE's design focuses on exhaustively enumerating small workloads, based on an empirical study of historical crash-consistency bugs in block-based file systems that showed most could be reproduced with at most three operations. It was unclear whether this would hold for PM file systems. However, 15 of the 18 bugs we found in PM file systems can be found using ACE with at most three operations, suggesting that this same *small-scope hypothesis* [23] holds for PM file systems. To try to invalidate this hypothesis, we also run FlyTrap using the Syzkaller gray-box fuzzer, which can generate much longer workloads but without the exhaustiveness guarantees of ACE (§3.4). Syzkaller found 3 bugs that ACE did not. However, all three bugs were found on short workloads: two would be considered seq-2 and one seq-3 in terms of the number of core system calls required. ACE missed them not because of size but because of complexities that ACE omits to make exhaustive enumeration tractable, such as testing unaligned writes.

Observation 7: Most of the observed buggy crash states involve few writes. FlyTrap generates crash states by snapshotting known-persistent disk states between store fences, and then replaying all subsets of the in-flight writes between each store fence (§3). For a system call with n in-flight writes before a fence, this means FlyTrap should consider all $2^n - 1$ possible crash states. However, we found that most bugs found by FlyTrap involve crash states that include *small* subsets of the in-flight writes. Of the 11 bugs in Table 2 that involve a crash in the middle of a system call, 10 can be exposed by a crash state that replays only a *single* write onto the last known-persistent state; the final bug requires two writes. This observation suggests a profitable heuristic for rapid crash-consistency testing would be to only test small subsets of in-flight rights rather than all of them. FlyTrap exploits

this observation by enumerating crash states in increasing order of subset size, allowing it to find most crash-consistency bugs quickly. In our experiments, we often cap the number of writes that are replayed to build each crash state, primarily to prevent Syzkaller from spending many hours checking a single outlier test that has a high in-flight write count. The highest in-flight write count we observed, 20 writes in some PMFS `write()` calls, would take about 30 hours to check exhaustively using FLYTRAP. A cap of two is enough to find all bugs presented in this paper; a cap of five is sufficient to check all crash states for most system calls in the PM file systems we tested.

5.2 Lessons Learned

Based on our observations above, we have distilled three lessons for developers of PM file systems and for building the testing tools that support them.

Lesson 1: Synchronous crash consistency on PM file systems simplifies the user experience, but complicates implementation and testing. Crash-consistency guarantees in modern file systems are something of a vicious cycle. File-system developers argue that relaxed guarantees are required to extract reasonable performance [30], but these weak guarantees are a pain point for application developers and have caused severe data loss in popular applications [9, 15, 40], so file-system developers implement workarounds to "fix" common mistaken application patterns and make the intended guarantees even less clear. The fine write granularity and low latency of persistent memory finally offers a path to strengthen file-system crash-consistency models, making resilient applications easier to build and validate. PM file system developers have taken advantage of this opportunity by making all system calls synchronous and durable.

While this end result is exciting, implementing it correctly carries new risks for PM file-system developers compared to traditional file systems. We found that many PM file-system bugs come from complex optimizations to realize high-performance synchronous crash-consistency—combining in-place updates with other consistency mechanisms, and replacing persistent state with reconstructible volatile state—that are uncommon techniques on slower storage media. This is a rich new design space for storage systems, and identifying the right primitives for this optimizations will be good future work. These optimizations also create complexity for testing and validation of PM file systems, which we found requires driving the file system into exercising deeper data structure manipulations and recovery mechanisms than existing crash-consistency tools are capable of.

Lesson 2: Diverse testing mechanisms and checkers help invalidate assumptions about crash-consistency patterns. Most crash-consistency testing tools build on heuristics and patterns in historic bugs to select the workloads they test. We expected to bring those patterns across to PM file systems, focusing on short workloads and a small set of potential crash points and patterns. However, we found instead that most assumptions about file-system crash consistency do not carry across to persistent memory, where the consistency mechanisms and guarantees are significantly different. Finding crash consistency bugs in PM file systems requires exploring many more crash states than other file systems, including crashes in the middle of system calls; we had to develop new techniques to make this search tractable. Existing file-system crash-consistency testing tools would not have found these bugs. We also found that fuzzing was an effective way to invalidate assumptions we had brought along from prior file systems experience, such as the significance of unaligned writes and exercising per-CPU code paths.

Another assumption we carried into this work was that the difficulty of building a PM file system lies in correctly applying the PM programming model. We intended to focus on exhaustively testing the precise persistency behavior of PM file system code. However, we found instead that most PM file system bugs were logic errors in optimizations that could also have occurred in block-based file systems (though likely would not improve performance in that context). Existing tools that focus on detecting specific PM programming error patterns [5, 16, 19, 21, 31–33, 39] would miss many of these bugs, but we were able to detect them even with small workloads and with very few in-flight writes applied. Writing general-purpose consistency checks and applying gray-box fuzzing to generate workloads helped to invalidate these assumptions and gave new insights into common bug patterns in PM file systems.

Lesson 3: Lightweight testing offers a scalable approach to detecting many crash-consistency bugs. FLYTRAP is, in principle, a bounded exhaustive testing [38] (i.e., bounded verification) tool for PM file systems: given enough time, it can check every possible crash behavior of every possible workload up to some bounds on its size and inputs. Of course, it is not tractable to exhaust this search space even with very small bounds. However, we found that FLYTRAP is an effective *lightweight* testing tool, in that it can quickly and automatically find many bugs by checking small workloads and few crash states, and then run for longer to find more corner-case issues. FLYTRAP runs the ACE seq-1 workloads on a file system in under 15 minutes and, when considering only crash states with one persisted write, can detect 12 of the 18 bugs we discovered. On the other hand, the fuzzer frontend to FLYTRAP takes 1–2 orders of magnitude longer to run (i.e., runs overnight) but finds three more bugs than ACE. These two frontends are complementary. They enable a lightweight approach that helps developers iterate quickly on new code, while still offering stronger confidence as the code gets "closer to production" [10].

6 Conclusion

This paper presents FLYTRAP, a new record-and-replay framework for testing the crash consistency of PM file systems. We use FLYTRAP with the ACE workload generator and the Syzkaller gray-box fuzzer and find 18 unique bugs across four PM file systems. To the best of our knowledge, this is the largest corpus of crash-consistency bugs on PM file systems. Our study of these bugs provides insights into how crash-consistency bugs arise in PM file systems and what types of tools are needed to test these systems.

References

[1] Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[2] Intel Optane DC Persistent Memory Quick Start Guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf.

[3] Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[4] Kernel Probes (Kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt.

[5] Discover Persistent Memory Programming Errors with Pmemcheck. https://www.intel.com/content/www/us/en/developer/articles/technical/discover-persistent-memory-programming-errors-with-pmemcheck.html, 2018.

[6] The Open Group Base Specifications Issue 7. https://pubs.opengroup.org/onlinepubs/9699919799/, 2018.

[7] Syzkaller. https://github.com/google/syzkaller/, 2021.

[8] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020.

[9] Nicolas Boichat. Issue 502898: ext4: Filesystem corruption on panic, June 2015. https://code.google.com/p/chromium/issues/detail?id=502898.

[10] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 836–850, October 2021.

[11] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In

Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, pages 83–98, Atlanta, GA, USA, April 2016.

[12] Vijay Chidambaram. *Orderless and Eventually Durable File Systems*. PhD thesis, University of Wisconsin, Madison, Aug 2015.

[13] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 85–95, New York, NY, USA, 2005. Association for Computing Machinery.

[14] 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 ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[15] Jonathan Corbet. ext4 and data loss, March 2009. http://lwn.net/Articles/322823/.

[16] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 503–516, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the ZoFS user-space NVM file system. *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[18] 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 Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[19] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 100–115, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[21] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 415–428, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[23] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng.*, 22(7):484–495, July 1996.

[24] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. WineFS: A hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 804–818, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

[26] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[27] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147–161, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association.

[30] Bug 15910 - zero-length files and performance degradation, 2010. https://bugzilla.kernel.org/show_bug.cgi?id=15910.

[31] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. Pmfuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 487–502, New York, NY, USA, 2021. Association for Computing Machinery.

[32] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.

[33] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[34] R. McMillan. Amazon Blames Generators For Blackout That Crushed Netflix. http://www.wired.com/wiredenterprise/2012/07/amazonexplains/, 2012.

[35] R. Miller. Power Outage Hits London Data Center. http://www.datacenterknowledge.com/archives/2012/07/10/power-outage-hits-london-data-center/, 2012.

[36] R. Miller. Data Center Outage Cited In Visa Downtime Across Canada. http://www.datacenterknowledge.com/archives/2013/01/28/data-center-outage-cited-in-visa-downtime-across-canada/, 2013.

[37] R. Miller. Power Outage Knocks Dreamhost Customers Offline. http://www.datacenterknowledge.com/archives/2013/03/20/power-outage-knocks-dreamhost-customers-offline/, 2013.

[38] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey and ACE: Systematically testing file-system crash consistency. *ACM Trans. Storage*, 15(2), apr 2019.

[39] 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.

[40] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[41] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash consistency. *Commun. ACM*, 58(10):46–51, 2015.

[42] J. Verge. Internap Data Center Outage Takes Down Livestream And Stackexchange. http://www.datacenterknowledge.com/archives/2014/05/16/internap-data-center-outage-takes-livestream-stackexchange/, 2014.

[43] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[44] R. S. V Wolffradt. Fire In Your Data Center: No Power, No Access, Now What? http://www.govtech.com/state/Fire-in-your-Data-Center-No-Power-No-Access-Now-What.html, 2014.

[45] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage*

Technologies (FAST 16), pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[46] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 478–496, New York, NY, USA, 2017. Association for Computing Machinery.

[47] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182, 2020.