Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems

Hayley LeBlanc
University of Texas at Austin

Shankara Pailoor
University of Texas at Austin

Om Saran K R E
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 Chipmunk, a new framework to test persistent-memory (PM) file systems for crash-consistency bugs. Using Chipmunk, we discovered 23 new bugs across five PM file systems; most bugs have been confirmed and fixed by developers. The discovered bugs have serious consequences, including making the file system un-mountable or breaking rename atomicity. We present a detailed study of the bugs found using Chipmunk and discuss important lessons learned for designing and testing PM file systems.

CCS Concepts: • **Software and its engineering → File systems management**; • **General and reference → Reliability**; • **Information systems → Storage class memory**.

Keywords: Crash consistency, file systems, persistent memory, testing, bugs

ACM Reference Format:
Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. 2023. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3567498

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 [1, 2]. PM has been a focus of research over the past two decades [3–5], and more recently has been commercialized by Intel [6]. This work has led to the development of variety of user space applications, libraries, and tools for PM [3, 4, 7–16]. A number of file systems [3, 16–23] have also been developed that exploit PM's unique properties.

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 power loss or a kernel bug. To do so, the file system should be *crash consistent*: it should recover after a failure to a consistent state without losing data the user expected to be persistent [24–27]. The crash-consistency specifications of disk-based file systems are based on the use of `fsync` and related system calls that flush the volatile page cache to persistent storage. If a file has not been explicitly flushed, its expected state after a crash is not defined. Several tools [28–30] are designed to test disk-based file systems for crash-consistency bugs, but all rely on the inclusion of `fsync` or related calls to establish crash-consistency guarantees.

Work on PM file systems has produced a diverse set of systems with new crash-consistency specifications and architectures, including in-kernel [17, 18, 20, 31], kernel-bypass [21–23], and hybrid systems [19]. A key defining trait of these systems is that most updates to persistent data bypass the page cache and write directly to underlying storage media. Also, unlike disk-based Linux file systems, which use a common kernel block layer to issue writes, these systems access PM directly using memory loads and stores. Most PM file systems take advantage of the media's low latency to guarantee atomic, synchronous operations without requiring `fsync`. The modified storage stack and strengthened crash-consistency guarantees makes existing file-system testing tools incompatible with PM file systems.

This paper makes two contributions. First, it presents Chipmunk, a framework for testing the crash consistency of any PM file system that implements the POSIX interface. Given a workload and a target file system, Chipmunk simulates crashes at points in the workload where some data is expected to be persistent. It then mounts the file system on resulting *crash states* and checks if it recovers correctly. The crash-consistency guarantees provided by various PM file systems are encoded into Chipmunk's consistency checker, enabling it to detect both low-level PM programming errors and high-level logic bugs that impact crash consistency. We couple Chipmunk with ACE [28], a tool that systematically generates small workloads, and Syzkaller [32], a gray-box

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567498

fuzzer. The test programs generated by these tools enable Chipmunk to be used for lightweight checks during development as well as for more thorough, long-running testing.

Second, this paper presents an analysis of 23 bugs found by Chipmunk across five PM file systems. We have reported all bugs upstream; all except those found in PMFS (which is not actively maintained) have been acknowledged, and 16 have been fixed. The bugs have severe consequences such as breaking the atomicity of the `rename` system call, which many applications depend on for atomic updates. To the best of our knowledge, this is the largest published corpus of PM file-system crash-consistency bugs. We analyze these bugs to distill useful insights for both PM file-system design and efficient crash-consistency testing of PM file systems. For example, many bugs are logic or design issues in performance optimizations rather than the missing flush/fence bugs that PM bug-finding tools target [33–40].

Challenges with testing PM file systems. Current approaches for testing file-system crash consistency do not work on PM file systems for two reasons. The first challenge is intercepting writes to the storage media. Past work on testing disk-based systems takes a black-box approach by recording writes as they go through the kernel block layer [28, 29]. However, PM file systems write directly to media using processor store instructions. A black-box approach to intercepting these writes would require instrumenting and recording all individual memory stores to PM, which is prohibitively expensive.

Second, prior work injects crashes only after `fsync`-related system calls, as these are the only points at which the file system makes consistency guarantees. However, many PM file systems perform updates to durable storage synchronously and do not mandate `fsync` for consistency. In these systems, the effect of each system call becomes persistent by the time that system call returns. Furthermore, prior tools made a conscious decision to only inject crashes after a system call completes; they do not test what happens when you crash in the middle of a system call. While this decision helps focus testing on mature file systems, injecting crashes in the middle of system calls is crucial for new and complex PM file systems. The increased number of crash points, together with the fine granularity of PM I/O, threatens to make testing intractable.

These challenges are demonstrated by Yat [41], a hypervisor-based brute-force testing tool for Intel's PMFS file system [18]. Yat records individual memory stores and has limited optimizations to focus on interesting crash states. Its authors report that it would take over 5 years to fully check one of their three test workloads.

Chipmunk. Chipmunk tackles these challenges by exploiting insights about the way that PM file systems are built to intercept writes and implement a new testing strategy that targets interesting crash states without exhaustively testing all of them. In contrast to existing black-box approaches to checking traditional file systems, our approach is based on gray-box *function-level interception* to efficiently record PM I/O and to facilitate reasoning about which crash states are interesting to check. We observe that PM file systems use *centralized persistence functions* to perform writes to PM whenever data must become durable. Chipmunk uses Kprobes [42] and Uprobes [43] to intercept these functions and record I/O without modifying the file system. Intercepting writes at the function level rather than the instruction level greatly reduces overhead by reducing the total number of writes that are intercepted individually. This gray-box instrumentation uses minimal information about the source code to making logging feasible and efficient.

To choose which crash states to test in systems with strong crash-consistency guarantees, 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 stores when appropriate. For example, a 1KB write to a file causes 128 8-byte writes (the unit of write atomicity on Intel PM), which would result in 2^{128} crash states to test. Even if we combine these writes into cache lines, there are still thousands of crash states to check. However, checking all of these states is unlikely to expose more bugs than just checking a few. We also observe that the set of in-flight writes (writes in volatile caches that have not yet become persistent) at any point in metadata-related system calls is typically small, reducing the number of crash states that Chipmunk must explore.

Generating workloads. Given a workload, Chipmunk provides a mechanism to generate and test crash states. An orthogonal question is deciding which workloads to explore. CrashMonkey [28] 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 used by CrashMonkey for use with synchronous PM file systems. To try to invalidate this hypothesis, we also use the Syzkaller [32] gray-box kernel fuzzer to generate more complex workloads.

Results. We use Chipmunk to test seven open-source PM file systems. This set includes two disk-based systems ported to PM, four in-kernel systems built for PM, and one hybrid file system with both user and kernel components. Chipmunk finds 23 unique bugs across five of the systems. From these results, we draw some common observations about how PM file systems work and how to test them, including:

- While a number of recent tools focus on finding missing or duplicate cache flushes and fences in PM applications [33–40], we found that a majority of the bugs (19/23) resulted from *logic errors*, such as a metadata item being left out of a transaction.

- PM file systems increase performance by maintaining some data structures in volatile DRAM and rebuilding them when the file system is remounted [17, 19, 20]; Chipmunk 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, instead of inside a transaction.

Our analysis of these bugs contain several 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 widely-applicable tools to test crash-consistency of PM file systems (§3);
- A corpus of 23 crash-consistency bugs discovered by these tools across five PM file systems (§5);
- An analysis of discovered crash-consistency bugs, with insights for PM file-system design and crash-consistency testing (§5).

Chipmunk is publicly available at https://github.com/utsaslab/chipmunk.

2 Background and Motivation

This section first 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 to a power loss or a kernel bug [24–26]. 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 [44].

Many applications depend on the file system to be crash consistent [45]. 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 [46–51]. 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 [6, 52], 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.

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 the caches 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 event of a crash. If there are multiple in-flight writes to different addresses, they may become persistent in any order.

PM file systems. Research on PM file systems has produced a variety of new systems that take advantage of PM's unique characteristics. PMFS [18], NOVA [17], NOVA-Fortis [31], and WineFS [20] are implemented in the kernel. Strata [21], Assise [23], and SplitFS [19] are implemented as kernel-bypass systems. Strata and Assise are implemented entirely in user space, while SplitFS handles file data in user space and passes metadata operations to a kernel component. Several systems (ext4-DAX and XFS-DAX [53]) are based on existing disk-based file systems. These systems share much of their code with their original implementations.

PM file systems differ from traditional file systems in several key ways. First, traditional file systems write updates to a volatile page cache in DRAM before flushing to disk. In contrast, PM file systems synchronously write some (if not all) updates directly to storage media in order to take advantage of the low latency and high bandwidth of PM. Second, while traditional systems make use of a common kernel-level block layer to issue writes to disk, PM file systems perform I/O directly using memory loads and stores without an extra software layer. Third, most PM file systems do not require use of `fsync`-related system calls to ensure that data becomes durable; the exception to this is ext4-DAX and XFS-DAX, which retain the crash-consistency properties of ext4 and XFS. We refer to systems that do not require `fsync` as having *strong* crash-consistency guarantees, whereas ext4-DAX and XFS-DAX have *weak* guarantees.

Why current tools are not enough. Existing work on crash-consistency testing (§6) is insufficient for today's PM file systems for four reasons. First, prior work on testing disk-based file systems cannot record writes to PM. CrashMonkey [28] and Hydra [29], two state-of-the-art tools for testing traditional file systems, rely on the kernel block layer to record disk I/O. Since PM file systems do not use the block

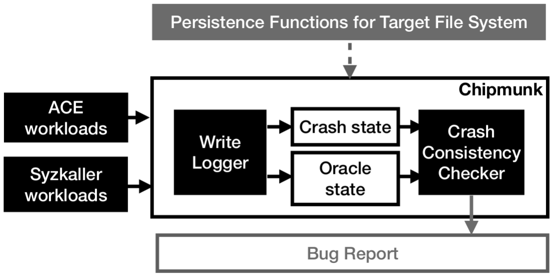

Figure 1. Architecture. Given a target file system and its persistence functions, Chipmunk uses workloads from both ACE and Syzkaller to test the file system. Chipmunk produces bug reports with enough detail to reproduce the bug.

layer, these tools are incapable of intercepting writes made by these systems. Second, these tools do not check all necessary crash states. CrashMonkey and Hydra only insert crashes after `fsync`-related system calls. Injecting crashes during system calls is crucial for exposing bugs in the complex and untested crash-consistency mechanisms of PM file systems. Furthermore, the consistency checkers for these tools would need to be rewritten to properly check these crash states.

Third, tools for testing PM file systems do not scale well. Yat [41], PMTest [33], and Vinter [54] record individual PM I/O instructions, resulting in a high number of instrumentation points. Vinter uses PANDA [55] for dynamic binary instrumentation, which introduces significant overhead. Yat has limited support for state space reduction and brute-force checks a large number of crash states.

Fourth, prior work on testing general PM applications cannot test high-level crash-consistency properties of file systems. These tools focus on *PM programming errors*, like missing or unnecessary cache-line flushes and store fences [33–37, 40]. Several tools [38, 39] can detect narrow classes of logic bugs – for example, that certain fine-grained updates are atomic – with hard-coded checks or developer-provided oracles. We are interested in checking higher-level crash consistency guarantees without requiring specifications from developers, so these tools are not sufficient.

3 Chipmunk

We present Chipmunk, a new framework to find crash consistency bugs in PM file systems. Chipmunk tackles the challenges outlined in §2 with function-level interception and a new testing strategy tailored to PM file systems. Chipmunk can test all PM file systems implementing POSIX, and requires no modification of file system code. We have run Chipmunk on file systems in both user and kernel space.

3.1 Overview

Chipmunk 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. Workloads are run sequentially, so there is only one system call running on the file system at any given time. It then replays recorded writes to create crash images, which represent the state of the system if it had crashed at different points during the workload. Chipmunk mounts the target file system on the crash image, lets it recover, and then checks whether it has recovered to a consistent state.

We use two tools to generate workloads for Chipmunk to test. The ACE workload generator is based on a hypothesis from the CrashMonkey work [28] that testing small workloads on a newly-created file system is effective at finding crash-consistency bugs. To determine if this hypothesis holds for PM file systems, we also use the Syzkaller [32] gray-box fuzzer to generate long, complicated workloads.

We used Chipmunk to test seven file systems: six in-kernel systems and one hybrid system with both user and kernel components. To our knowledge, there are no publicly-available user-space PM file systems that support recovery from arbitrary crashes (§4.1).

3.2 Challenges

In order to effectively test PM file systems, Chipmunk must overcome three key challenges: how to intercept writes, how to deal with new crash-consistency semantics, and how to deal with very large sets of crash states. We describe each challenge and outline the empirical observations and design decisions that allow Chipmunk to discover many bugs in PM file systems.

Intercepting writes. Prior work on testing file-system crash consistency has taken a black-box approach to recording writes to storage media. Yat [41] uses a modified hypervisor that triggers a VM exit on stores, flushes, and fences to PM, PMTest [33] uses the tracking mechanism provided by WHISPER [15] to trace these instructions, and Vinter [54] uses dynamic binary instrumentation in PANDA [55]. These approaches introduce overheads associated with the instrumentation tools and a high number of instrumentation points.

CrashMonkey [28] and Hydra [29], two state-of-the-art tools for testing crash-consistency in traditional file systems, also use a black-box approach based on the Linux kernel's block layer. The file systems they target issue all writes to storage via this layer, so it provides a natural interception point. However, since PM file systems do not use this layer, we cannot use this approach to log writes in Chipmunk.

Instead, Chipmunk uses gray-box *function-level instrumentation* to intercept writes to PM. Each PM file system we examined uses a small set of *centralized persistence functions* to perform I/O. These abstractions simplify reasoning about PM semantics and potentially enable portability to new architectures. All tested systems implement functions for some subset of the following: non-temporal `memcpy`, non-temporal memset, flushing cache lines associated with a buffer, and issuing store fences. Each of these operations handles a single, contiguous non-temporal store, a contiguous set of cache line flushes, or enforces store ordering. Chipmunk is not limited to recording just this set of functions; a system's logger can be written to handle other types of persistence functions, if, for example, the system is designed for another persistence model.

Chipmunk requires developers to provide the names of centralized persistence functions functions; it then instruments these functions at runtime using the Kprobes [42] and Uprobes [43] debugging mechanisms in the Linux kernel. This gray-box approach to recording writes has multiple benefits. It makes logging feasible without requiring source code modification, and it makes Chipmunk portable to new PM architectures since the semantics of x86 PM primitives are not built into the recording code. It also enables Chipmunk to encode information about the context in which a write was made and use it during consistency checking.

New crash-consistency semantics. In traditional file systems, if a user wants to ensure that a specific file or set of files is persistent on disk, they must call an fsync-related system call to flush updates from the volatile page cache to the storage media. Since crash-consistency guarantees are not well defined if the system crashes prior to such system calls, and the journaling mechanisms that handle incomplete updates are very mature, systems like CrashMonkey and Hydra only insert crash points after fsync-related calls.

However, PM file systems with strong crash consistency specs clearly define the correct state of each file at every point during execution, not just after fsync. These systems guarantee that most operations are both synchronous and atomic. To test the novel, complex, and un-tested crash-consistency mechanisms of PM file systems, Chipmunk must inject crashes during system calls (and not just after fsync). We developed a new testing strategy and set of consistency checks (§3.3) to handle these new crash points for systems with strong guarantees. Chipmunk coalesces logically-related non-temporal stores and flushes (e.g., those all associated with the same file data write) and replays them in different combinations to focus on interesting crash states. It uses an oracle-based checker that compares the post-crash state of each file to a set of possible legal states.

Increased number of crash states. PM file systems with strong crash consistency make fine-grained writes to storage media in the critical path of system calls. As a result, a workload can result in significantly more crash states that are interesting to test than in systems with weaker guarantees. Specifically, if the number of in-flight writes between each store fence is too high, the number of possible crash states to check will explode to an intractable number.

We studied five systems with strong guarantees and made two observations that we use to overcome this challenge. First, when PM file systems perform metadata operations, they issue a small number of small writes to PM with frequent store fences. We found that the average number of in-flight writes for metadata operations is three and the maximum is 10 in the tested systems. This means that at any given crash point in metadata-related system calls, the number of crash states is small enough to test exhaustively. Second, although file data operations often involve many in-flight writes with few store fences, checking every possible crash state is unlikely to expose new bugs. We want to check how the file system recovers when *some* file data is lost in a crash, but it is unnecessary to check every possible state. Chipmunk coalesces data associated with the same file data update into a single write, and checks only a small subset of states with missing data. Metadata about the size of buffers and how they are written guides this heuristic; for example, non-temporal memcpy on a large buffer usually indicates a file data write.

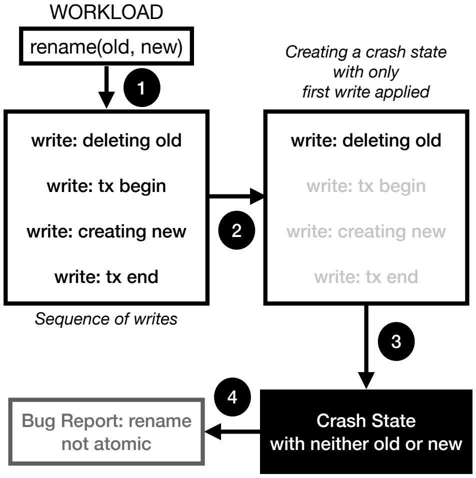

Figure 2. Chipmunk workflow. The figure shows how crash consistency is tested using a rename() workload. In this example, the old file being deleted is updated in-place, while the new file creation happens inside a transaction. 1) Chipmunk runs the workload and logs a sequence of PM operations. For simplicity, the operation has been reduced to a sequence of four logical writes. 2) Chipmunk creates a crash state where only the old file is deleted; the other writes are lost. 3) The consistency checker finds that both the old and new files are missing. 4) Chipmunk creates a bug report. Chipmunk discovered this bug in NOVA (bug 4).

3.3 Chipmunk Architecture

Chipmunk is built on top of the CrashMonkey framework [28]. CrashMonkey consists of a set of user space utilities, a block device wrapper kernel module that intercepts writes, and a copy-on-write device to facilitate constructing file system snapshots. We adapt CrashMonkey's user space utilities to target PM file systems and build new kernel modules based on Kprobes and Uprobes, two Linux kernel debugging utilities, to record writes to PM.

Given a workload and a target file system, Chipmunk proceeds in four 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 (4) generate a bug report if required. We now describe these steps in detail.

Logging writes. Chipmunk uses two dynamic debugging and tracing tools, Kprobes [42] (for in-kernel file system components) and Uprobes [43] (for user space components), to automatically instrument centralized persistence functions at runtime. Our logging modules only require the name (for kernel space components) or offset (for user space object files) of these functions in order to instrument them. Kprobes and Uprobes are used together in the same logging module to test SplitFS.

Kprobes and Uprobes make a copy of a probed instruction and replace the first byte(s) with a breakpoint instruction. When the breakpoint is hit, control is passed to a handler function. In Chipmunk's loggers, these handlers record the probed operation and its arguments. The destination of each update is translated from a virtual to physical address within the logging module to facilitate later replay. The user-space test harness also inserts markers into this log to record the start and end of each system call, which Chipmunk uses to determine which writes to PM are associated with 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 since these functions are used extremely frequently.

Constructing crash states. Given a workload and a file system to test, Chipmunk selects crash points based on the crash consistency guarantees of the file system and simulates crashes at these points. For ext4-DAX and XFS-DAX, crash points are placed after `fsync`, `sync`, and `fdatasync` calls. For the other systems, crash points are injected after each store fence invoked by the file system. A single system call may perform multiple store fences, resulting in multiple crash points per system call. We construct and create possible crash states out of the in-flight writes that follow each store fence. This process checks crash states that occur both during and between system calls.

Chipmunk replays a workload by walking through the log of flushes, non-temporal stores, and fences. When it encounters a cache line flush or non-temporal store, it adds a structure containing the type, contents, and destination address of the flush/store to an in-flight vector. When it encounters a store fence, it first constructs and checks crash states based on the in-flight vector, then flushes the contents of the vector to the replay device. Each crash state is constructed by replaying a subset of the contents of the in-flight vector on top of all updates that precede the most recent store fence, which are guaranteed to be persistent. We replay the updates in each subset in program order. For n in-flight writes, there will be $2^n - 1$ crash states to check. As noted in §3.2, we have observed that n is small in practice for metadata-related calls, allowing Chipmunk to apply this exhaustive testing strategy. File data writes do not adhere to this pattern but can be coalesced into a small set of large writes. Since a small number of in-flight writes is not a guarantee, Chipmunk 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 crash states. To check file-system consistency, Chipmunk first mounts the target file system on each crash state, which is itself a useful consistency check. Once successfully mounted, Chipmunk compares the file-system state against an oracle representing valid post-crash state(s). The oracle runs the original workload on a fresh file system instance in parallel with log replay. When Chipmunk encounters the beginning of a new system call in the log, it records the current oracle state of file(s) that will be modified, then executes that system call on the oracle file system. Files in a crash state are compared against an oracle version of the file by checking whether metadata provided by `stat` differs between the two versions. For regular files, Chipmunk also compares the contents of each version. For directories, Chipmunk compares the directory entries of each version. Several crash states may be compared to the same few oracle states, so Chipmunk caches the metadata and contents for each oracle file version in memory.

Most system calls in file systems with strong guarantees are intended to be atomic. The main exception is `write`, although many systems provide the option to make `write` atomic. Further, all system calls are *synchronous*, in that modifications to persistent data are made durable by the time each system call completes. Chipmunk focuses on checking these atomicity and synchrony properties.

When a crash is injected in the middle of a system call, Chipmunk checks that the operation is atomic by comparing modified files in the crash state to the current and previous oracle versions. If the operation modifies multiple files, the files must all match the same version. When a crash is injected after a system call, Chipmunk checks that the system call is synchronous by comparing the crash state against the current oracle state. Chipmunk also confirms that files that should be unmodified by the current system call match the current oracle state in each crash state. These checks validate properties implied by POSIX or widely expected by users in practice [27, 45]. Finally, to validate that the file system is in a usable state, Chipmunk creates files in all directories, then deletes all files. If any of these checks or operations fail, Chipmunk outputs a bug report describing the inconsistency and the corresponding crash state.

Because the consistency checks mutate the crash 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, Chipmunk generates crash states and tests them for consistency. An orthogonal challenge is generating workloads for Chipmunk to test. The CrashMonkey work [28] introduced the hypothesis that small workloads on new file systems are useful in finding crash-consistency bugs. While this hypothesis was true on traditional 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 Chipmunk. We also modify the Syzkaller [32] gray-box fuzzer to work with Chipmunk. 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 [28] to systematically generate workloads. ACE was designed to exhaustively generate workloads of a pre-defined structure to test 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 satisfies dependencies by opening and closing files and adds `fsync`, `fdatasync`, or `sync` operations. A workload with *n* core system calls is called a "seq-*n*" workload.

We use a slightly modified version of ACE's default mode, which inserts at least one `fsync`-family operation in each workload, for ext4-DAX and XFS-DAX. We also added a mode that does not insert these calls for the other systems.

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. "seq-3 metadata" workloads [28]) to make testing tractable. For PM file systems with strong consistency, we generate 56 seq-1 tests, 3136 seq-2 tests, and 50650 seq-3 metadata tests. The default mode generates 419 seq-1 tests and 432462 seq-2 tests; we did not run seq-3 metadata tests on ext4-DAX and XFS-DAX.

3.4.2 Syzkaller. We modify Syzkaller [32], a state-of-the-art gray-box kernel fuzzer, to generate workloads for Chipmunk. 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 Chipmunk from those seeds. As Chipmunk executes each workload, Syzkaller collects code coverage information by recording coverage points inserted by the compiler. 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 system calls and 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 [56] for the call's arguments. For example, the template for `write` specifies that its first argument is a valid file descriptor in use by the program, rather than an arbitrary integer.

To adapt Syzkaller to our setting, we restrict it to only generate workloads that contain file-system operations, and replace its workload executor with a custom one. Our executor invokes Chipmunk on each workload and records code coverage both before the crash and during recovery. For PM file systems with strong consistency, we add crash points between each system call and in the middle of the final non-failing system call in the workload. For ext4-DAX and XFS-DAX, we include `fsync`, `sync`, and `fdatasync` in workloads and check crash states after each call to one of these system calls. We add a `sync` at the end of each workload to make sure we check at least one crash state. Since Syzkaller is a kernel fuzzer and we are primarily interested in collecting coverage on SplitFS's user space component, we use GCC's sanitizer coverage instrumentation to collect code coverage [57] and modify Syzkaller to use this coverage information. This required adding some code to SplitFS to log the basic blocks covered during fuzzing, but did not require modification of any existing code. We do not collect code coverage of SplitFS's kernel component.

Like many fuzzers, Syzkaller can quickly generate many bug reports that are duplicates. In our setting, this duplication also arises when multiple crash states trigger the same bug. To address this problem, we extended Syzkaller to automatically triage bug reports generated by Chipmunk during fuzzing. We use a simple triaging procedure that clusters bug reports by lexical similarity. We also updated Syzkaller to display these bug report clusters in its UI dashboard to make them easier for users to debug.

3.5 Implementation

Although Chipmunk was originally based on CrashMonkey, the two systems diverged early in development and most of Chipmunk's core code is new. Chipmunk has twice as much code dedicated to constructing and checking crash states as CrashMonkey. The increase in test harness size primarily comes from the complexity of PM write semantics when constructing crash states, the need to associate each logged write operation with the system call that issued it, and a much more complex set of consistency checks that account for the semantics of each tested system call in different file systems. We also wrote new code to track oracle state for the consistency checks, as CrashMonkey's was insufficient for checks outside of `fsync`-related calls. Running SplitFS also required further modification to be made to the test harness, since it requires its object files to be dynamically linked to the test harness at runtime. About 2000 LOC in Chipmunk's core testing infrastructure comes from a Syzkaller-specific test harness that executes fuzzer-generated tests. The only parts of CrashMonkey that remained largely unchanged were the code that loads and runs ACE-generated tests and some functions related to recording the system calls in a workload.

Chipmunk's core testing infrastructure consists of about 9000 lines of C++ code as reported by `sloccount`. Five system-specific logger modules add about 1000 lines of C code each (several similar systems share modules). We also added about 1000 lines of Go to Syzkaller to handle crash consistency tests and to collect coverage when remounting crash states.

3.6 Discussion

Limitations. We made certain design decisions to make testing PM file systems for crash consistency tractable. However, it is possible that these choices may cause Chipmunk to miss some bugs or limit its ability to test some file systems. First, Chipmunk cannot test every workload, and does not test all possible crash states for each workload. Second, Chipmunk assumes that the PM file system has centralized persistence functions. A PM file system that uses in-line assembly or macros to update PM would not be compatible with Chipmunk. Third, Chipmunk does not currently support checking concurrent workloads. Testing crash consistency for concurrent workloads is a hard problem [58, 59] that prior tools do not support. While supporting concurrent workloads could expose more bugs, it is out of the scope of this paper and we leave it as future work.

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

Persistence models. Chipmunk is implemented for x86's epoch-based persistence model, which at the time of writing was available on Intel's Optane DC Persistent Memory Module. Since then, Intel has cancelled their Optane project. However, other hardware vendors have announced similar byte-addressable persistent storage devices [60], which may use different persistence models [15, 61–63].

Because Chipmunk logs writes at the function level and runs checks on real file-system images, its techniques are not tied to any persistence model. Adding support for a new model would involve implementing its semantics in Chipmunk's replay logic and logger modules, but the rest of the framework would remain the same. Since the high-level operations governed by primitives in different persistence models are broadly similar, these changes should be straightforward. We expect that file systems for new models will use centralized persistence functions as the preferred abstraction for writing data to durable storage for portability and simplicity.

Furthermore, our bug analysis shows that many bugs are caused by logic errors rather than PM programming errors. Logic bugs will continue to occur across persistence models and we expect Chipmunk would be a valuable tool for testing file systems built for a variety of persistence models.

Code coverage. Although Chipmunk is not intended to be complete and achieving high code coverage metrics was not a goal of this work, our results indicate that using ACE and Syzkaller to generate workloads enables thorough testing of important file system features. Our workload generation strategy, which focuses on testing common file system operations, was effective at exposing many new crash-consistency bugs in the tested systems. For long running tests, Chipmunk could be paired with tools like OSS-Fuzz [64] to focus on high code coverage.

4 Testing PM File Systems

In this section, we evaluate Chipmunk's effectiveness at finding bugs across different PM file systems.

4.1 Methodology

File systems. We ran Chipmunk with seven open-source PM file systems: NOVA [17], NOVA-Fortis [31], PMFS [18], WineFS [20], SplitFS [19], and ext4-DAX and XFS-DAX [53]. NOVA, NOVA-Fortis, PMFS, WineFS, and SplitFS in strict mode have strong crash-consistency guarantees, so Chipmunk inserts crash points both during and after system calls when testing these systems. ext4-DAX and XFS-DAX have weak guarantees, so Chipmunk only inserts crash points after `fsync`-related system calls when testing them. Chipmunk is compatible with Strata and Assise as well, but we learned after communication with authors that neither system's current artifact supports recovery from arbitrary crashes, so we were unable to proceed with evaluation on these systems.

System calls. We select a set of system calls to test based on what is supported by each file system and what crash consistency guarantees they provide. We focused on ten key operations: `creat`, `mkdir`, `fallocate`, `write`, `link`, `unlink`, `remove`, `rename`, `truncate`, and `rmdir`. Tests run on ext4-DAX and XFS-DAX also include `setxattr` and `removexattr`, which are not supported by the other systems we tested. All tests also include `open` and `close` as necessary, and tests on ext4-DAX and XFS-DAX use at least one of `fsync`, `fdatasync`, or `sync` to ensure that data becomes persistent. We did not test `mmap` with Chipmunk, as modifications to memory-mapped regions are not handled via centralized persistence functions and a number of other tools have been built to target the crash-consistency of memory-mapped data (§6.3).

4.2 Experimental setup

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 (except for those testing SplitFS, which requires 32GB). 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. For the systems with strong crash consistency, we ran seq-1 and seq-2 tests on individual VMs, as the number of tests to run was relatively small. We split seq-3 metadata workloads across 10 VMs and ran them in parallel. At the time we ran these experiments, WineFS and SplitFS both had bugs that prevented many seq-3 tests from running. 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. For ext4-DAX and XFS-DAX, we ran seq-1 tests on an individual VM and split seq-2 tests across 20 VMs. We were unable to run seq-3 metadata tests on these systems due to time constraints.

To evaluate Chipmunk with Syzkaller, we ran seven Chameleon Cloud [65] 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 Chipmunk 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.

4.3 Evaluation

ACE tests. For each file system under test, Chipmunk took less than 1 hour to run seq-1 workloads on a single VM. Running these tests on NOVA/NOVA-Fortis, PMFS, and WineFS takes less than 15 minutes. Seq-2 tests take 7-20 hours on the PM file systems with strong consistency. It takes about 30 hours for all seq-2 tests to finish running on ext4-DAX and XFS-DAX when using 20 VMs in parallel. For systems tested on seq-3 workloads, it took 16-26 hours to run them in parallel on 10 VMs. 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, Chipmunk found 19 bugs using ACE tests across five of the tested systems.

Syzkaller. We ran Chipmunk with Syzkaller for 18 hours on 15 VMs, for a total of 270 CPU hours spent fuzzing each system. During this time, Chipmunk checked over 40 million crash states across all tested systems, finding 23 unique bugs. Four of these bugs cannot be found with ACE-generated workloads.

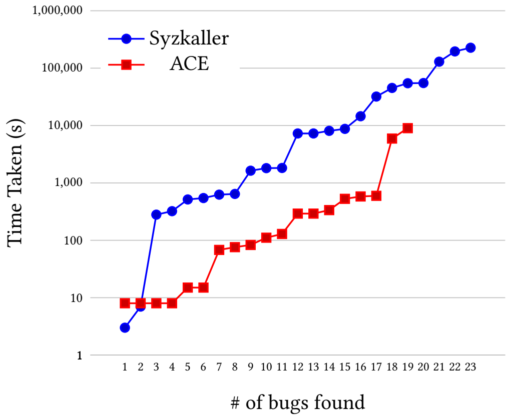

Figure 3. Cumulative time taken to find crash-consistency bugs by ACE and Syzkaller.

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 19 out of 23 bugs in less than three CPU hours total, but is unable to find the final four bugs. Aside from a couple bugs that Syzkaller and ACE workloads both trigger almost immediately, Syzkaller takes almost 20× more CPU time than ACE to find the first 12 bugs and almost 6× more CPU time to find all bugs exposed by ACE. However, when we let Syzkaller run for an additional 47 CPU hours, it is able to find four bugs that are not detected by ACE. ACE misses these 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. ACE workloads do not open multiple file descriptors for the same file and thus cannot trigger these bugs.

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 to find bugs during file-system development, whereas Syzkaller should be run for a long time in an environment with ample compute resources for more comprehensive crash-consistency testing.

4.4 Results

Crash-consistency bugs. Using ACE and Syzkaller generated tests, Chipmunk finds 23 new unique crash-consistency bugs across the tested file systems. The number of bugs is based on the number of unique fixes required to patch all of the bugs, not different user-visible consequences. Two bugs are found in both WineFS and PMFS for a total of 25 bugs.

Table 1 describes the consequences of each bug and the system calls they affect. The bugs are classified as either

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, unlink, rmdir, rename`	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
21	SplitFS	Operation is not synchronous	All metadata	Logic
22	SplitFS	File data lost	`write, pwrite`	Logic
23	SplitFS	File data lost	`write, pwrite`	Logic
24	SplitFS	Operation is not synchronous	All	Logic
25	SplitFS	Rename atomicity broken (old file still present)	`rename`	Logic

Table 1. Bugs found by Chipmunk, their consequences, and the system calls that they affect.

logic or PM errors (§5.1). Chipmunk found eight bugs in NOVA, five bugs in SplitFS, four bugs in NOVA-Fortis, two bugs in PMFS, two bugs in WineFS, and two bugs in both PMFS and WineFS. Many of these bugs have serious consequences: three prevent the file system from being mounted entirely, and three impact the atomicity of `rename`, which many applications rely on [45]. Many others cause data loss or prevent a user from accessing files entirely.

Bugs 4, 5, and 13 in Table 1 were found independently by both Vinter [54] and Chipmunk. Vinter's authors also reported a bug related to an optimization in the non-temporal store function used by NOVA and NOVA-Fortis, which Chipmunk can reproduce. Chipmunk found a related bug impacting PMFS and WineFS (17 and 20).

Chipmunk did not find any bugs in ext4-DAX or XFS-DAX. We attribute this to the maturity of the base file systems. Most code in ext4-DAX and XFS-DAX is shared with their non-DAX versions, which are very well tested. CrashMonkey also found no new bugs in either system, and Hydra found only one new crash-consistency bug in ext4.

Non-crash-consistency bugs. While working with Chipmunk, we also found eight non-crash-consistency bugs not included in Table 1. 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.

5 Bug Analysis

This section presents an analysis of the 23 crash-consistency bugs found by Chipmunk (Table 1). To the best of our knowledge, this is the largest corpus of crash-consistency bugs in PM file systems.

5.1 Observations

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

5.1.1 Nature of crash-consistency bugs.

Observation 1: Most of the observed bugs are logic issues rather than PM programming errors. Prior work on crash-testing PM applications focuses on bugs related to subtleties in the PM programming model, like CPU store reordering. However, the majority of bugs we found—19 of 23—are actually due to higher-level logic bugs rather than mistakes in managing PM. The "type" column in Table 1 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;

Observation	Associated bugs
Many bugs are logic/design issues, not PM programming errors.	1, 3–8, 10–13, 16, 19, 20, 21–25
The complexity of performing in-place updates leads to bugs.	4–7, 14, 15
Recovery related to rebuilding in-DRAM state is a significant source of bugs.	1, 3, 7, 11, 13, 16, 19, 24, 25
Complex features for increasing resilience can introduce crash consistency bugs.	2, 9–12
Many 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, 21–25
Many bugs are exposed by replaying a few small writes onto previously persistent state.	3–6, 9–13, 19, 20

Table 2. Observations and the bugs associated with them.

it must also check higher-level consistency properties that cannot be validated at the level of individual writes. We note that all bugs in found in SplitFS are logic bugs. Our results suggest that SplitFS's use of ext4-DAX to handle metadata operations reduces risk of PM programming errors, but does not eliminate logical bugs that impact crash consistency. All of the bugs Chipmunk found in SplitFS are related to its optimized logging approach, which SplitFS uses to provide stronger crash-consistency guarantees than ext4-DAX.

Observation 2: In-place update optimizations are a common source of crash consistency bugs. One of the allures of PM is that programs can access it as memory, performing fine-grained reads and writes directly rather than coalescing them into larger block-sized I/O operations. This design makes it possible in principle to reduce the overheads of traditional consistency mechanisms like journaling by manipulating on-disk data structures directly. Most of the systems we tested use a journal for crash consistency, but have performance optimizations to bypass the journal in certain circumstances. For example, NOVA updates the link count of a file by updating a per-inode log. Appending to this log is usually 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.

We found these optimizations to be particularly error-prone: six of 23 bugs in Table 1 are caused by in-place updates. For 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. 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). We tested both versions on Intel Optane DC Persistent Memory media. In a microbenchmark that repeatedly overwrites a file using `rename`, the fixed version 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. In contrast, PM file systems often keep metadata like free page lists in DRAM as a performance and write endurance optimization and rebuild them at mount. This rebuilding code is subtle because it must account for potential inconsistencies or partial states after a crash, and we found that nine of the 23 bugs in Table 1 were in such code. For example, bug 13 can be caused by a crash during a `truncate` system call on PMFS. This operation first stores information about the truncation 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 is kept in DRAM and thus lost in the crash. Attempts to replay truncations therefore cause a null pointer dereference.

Rebuilding volatile state is more complex in PM file systems that maintain *per-CPU* volatile state to improve scalability. For example, in bug 19, WineFS failed to properly index into an array of per-CPU journals that were read during crash recovery, preventing journaled updates from being accessed after a crash.

Observation 4: Resilience mechanisms to recover from media failures can introduce new crash-consistency bugs. NOVA-Fortis [31] is an extension of NOVA that adds fault detection and tolerance for media errors and software bugs by (among other techniques) replicating and/or checksumming inodes, logs, and file data. While NOVA-Fortis is not explicitly designed to increase crash resilience, we tested it to determine if it is more tolerant of crashes than NOVA.

NOVA-Fortis has all the same crash-consistency bugs we found in the original version of NOVA, and in addition has four 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: Many observed bugs require simulating crashes during system calls. Current crash-consistency testing tools for traditional file systems, like CrashMonkey [28] and Hydra [29], insert crashes only after `fsync`-related system calls. 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.

We found that this same heuristic does not work for PM file systems. 11 of the 23 bugs in Table 1 require a crash to occur during a system call. This is a corollary of our observation that most PM file systems implement most system calls synchronously, making their effects 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 the system call completes would hide these bugs, as NOVA flushes all writes by the end of the operation.

Observation 6: Short workloads suffice to expose many crash-consistency bugs. We use ACE [28] to exhaustively generate small test workloads. ACE's design is based on an empirical study of historical crash consistency bugs in traditional file systems that showed that most bugs could be reproduced with at most three core operations. It was unclear whether this would hold for PM file systems. However, 19 of the 23 bugs we found in PM file systems can be found using ACE, suggesting that this same *small-scope hypothesis* [66] holds for PM file systems. We also run Chipmunk using the Syzkaller gray-box fuzzer, which can generate much longer workloads but without the exhaustiveness guarantees of ACE (§3.4). Syzkaller found four bugs that ACE did not. However, all four bugs were found on short workloads: three would be considered seq-2 and one seq-3. ACE missed them not because of size but because of complexities that ACE omits to make exhaustive enumeration tractable, such as testing non-8-byte-aligned writes.

Observation 7: Most of the observed buggy crash states involve few writes to PM. Chipmunk 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 Chipmunk should consider all $2^n - 1$ possible crash states. However, we found that most bugs found by Chipmunk involve crash states that include *small* subsets of the in-flight writes. Of the 11 bugs in Table 1 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 would be to only test small subsets of in-flight writes. Chipmunk 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 with 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 Chipmunk. 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 [67], but these weak guarantees are a pain point for application developers and have caused severe data loss in popular applications [45, 68, 69], 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 PM 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, replacing persistent state with reconstructible volatile state, or introducing new logging protocols — that are uncommon techniques on slower storage media. This is a rich new design space for storage systems, and identifying the right primitives for these 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. However, we found instead that most assumptions about file-system crash consistency do not carry across to PM, 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. We also found that fuzzing was an effective way to invalidate assumptions 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. Existing tools that focus on detecting specific PM programming error patterns [33–40] would miss many of these bugs. Writing general-purpose consistency checks and applying gray-box fuzzing to generate workloads helped to invalidate these assumptions.

Lesson 3: Lightweight testing offers a scalable approach to detecting many crash-consistency bugs. Chipmunk is, in principle, a bounded exhaustive testing [28] 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 Chipmunk 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. Chipmunk runs the ACE seq-1 workloads in less than 15 minutes on most tested systems. On the other hand, the fuzzer frontend to Chipmunk takes 1–2 orders of magnitude longer to run but finds four more bugs than ACE. These two frontends are complementary. They enable a lightweight approach that helps developers iterate quickly on new code, while offering stronger confidence as the code gets "closer to production" [70].

6 Related work

This section discusses related work on crash consistency testing for PM and file systems.

6.1 Testing traditional file systems

CrashMonkey [28] is a black-box record-and-replay framework that uses systematically-generated test cases to find bugs in file systems for traditional storage media. Hydra [29] is a file-system fuzzer that focuses on crash consistency bugs and POSIX violations. Both CrashMonkey and Hydra rely on the kernel block layer to record write operations and are incompatible with PM file systems. Several model checking approaches [30, 71] require modifications to the kernel and file system. Further, all of these tools only check crash states immediately after `fsync`-related system calls, which is insufficient for finding bugs in the complex and untested crash-consistency mechanisms of PM file systems.

6.2 Testing PM file systems

Yat [41], PMTest [33], and Vinter [54] have all been used to test PM file systems for crash-consistency bugs. Yat was built for PMFS and records PM I/O using a custom hypervisor. Yat has limited optimizations to reduce the space of crash states. For example, the authors report that a workload of 1200 `creat`, `mkdir`, and `write` calls would take over five years to complete. PMTest was also only used on PMFS and found no crash consistency bugs.

Vinter [54] was developed concurrently with Chipmunk and is similar in many respects. Both tools use a record-and-replay approach and utilize strong crash-consistency guarantees to check specific, well-defined properties of crash states. Chipmunk is compatible with the ACE test generator and the Syzkaller kernel fuzzer, whereas Vinter was evaluated with 16 handwritten test cases. Vinter's authors report nine new bugs with five unique root causes in three file systems, whereas Chipmunk found 23 unique bugs in five file systems. Chipmunk can find all but one of the bugs reported by Vinter; the last bug only impacts file timestamps, which Chipmunk does not currently check.

To record PM I/O, Vinter instruments individual instructions using PANDA [55], which introduces significant overhead. As a result, Chipmunk is faster than Vinter: Vinter takes 24 minutes to run a suite of 16 tests on NOVA, whereas Chipmunk runs the ACE seq-1 suite in less than 15 minutes. To reduce the state space, Vinter focuses on crash states containing in-flight writes that are likely to be read during recovery. Chipmunk could incorporate this heuristic by recording PM read functions, but Vinter could not use Chipmunk's logical write coalescing heuristic because it does not have information about which function was used to write each buffer.

6.3 Testing PM applications

Recent research on PM crash consistency has focused on the difficulty of writing correct code against low-level PM programming models. These tools target PM programming mistakes and have limited support for identifying higher-level logic bugs. Many also require manual annotation of source code and do not currently support testing kernel code, severely limiting their applicability to file system testing.

Pmemcheck [35] is a Valgrind-based tool designed to find PM programming errors in applications built with PMDK [7]. Using Pmemcheck without PMDK requires manual annotation of source code. PMTest [33] and XFDetector [34] also require developers to manually annotate regions of interest. PMFuzz [36] is a fuzzer built on AFL++ that uses XFDetector and Pmemcheck to detect bugs.

Agamotto [38] is a symbolic execution tool built on KLEE for user-space PM applications. Agamotto does not require source code annotation, but finding bugs other than low-level PM programming errors requires developer-provided oracles. Witcher [39] is designed to test key-value stores and targets both PM programming errors and "persistence atomicity violations" by statically inferring which sequences of writes are intended to be atomic. PmDebugger [37] is a tool for collecting and analyzing PM access traces without source code annotation.

7 Conclusion

This paper presents Chipmunk, a new record-and-replay framework for testing the crash consistency of PM file systems. We use Chipmunk with the ACE workload generator and the Syzkaller gray-box fuzzer and find 23 unique bugs across five 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. Chipmunk is publicly available at https://github.com/utsaslab/chipmunk.

Acknowledgements

We thank our shepherd, Sanidhya Kashyap, and the anonymous reviewers at EuroSys '23 and OSDI '22, and the members of the Systems and Storage Lab at UT Austin for their insightful comments and suggestions. This work was supported by NSF CAREER #1751277, NSF SHF-CCF-1712067, the UT Austin-Portugal BigHPC project (POCI-01-0247-FEDER-045924), and donations from VMware and Amazon Web Services.

References

[1] 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.

[2] 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.

[3] 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.

[4] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[5] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, 2010.

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

[7] Intel Corporation. Persistent memory development kit. https://pmem.io/pmdk/.

[8] Intel Corporation. Redis. https://github.com/pmem/redis.

[9] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 897–912, Renton, WA, July 2019. USENIX Association.

[10] 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 '19)*, Ontario, Canada, October 2019.

[11] Nachshon Cohen, David T. Aksun, and James R. Larus. Object-oriented recovery for non-volatile memory. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[12] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 28–40, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, February 2015. USENIX Association.

[14] Mingzhe Zhang, King Tin Lam, Xin Yao, and Cho-Li Wang. Simpo: A scalable in-memory persistent object framework using nvram for reliable big data computing. *ACM Trans. Archit. Code Optim.*, 15(1), March 2018.

[15] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 135–148, New York, NY, USA, 2017. Association for Computing Machinery.

[16] 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.

[17] 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.

[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] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Junfeng Yang, Can Sar, and Dawson Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 131–146, USA, 2006. USENIX Association.

[31] 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.

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

[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] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] Uprobe-tracer: Uprobe-based event tracing. https://www.kernel.org/doc/html/latest/trace/uprobetracer.html.

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

[45] 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.

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

[47] 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.

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

[49] 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.

[50] 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.

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

[52] 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.

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

[54] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 933–950, Carlsbad, CA, July 2022. USENIX Association.

[55] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, New York, NY, USA, 2015. Association for Computing Machinery.

[56] 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.

[57] Program instrumentation options. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html.

[58] Madan Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, November 2007.

[59] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 415–431, Broomfield, CO, October 2014. USENIX Association.

[60] Samsung electronics introduces industry's first 512gb CXL memory module. https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module, 2022.

[61] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 265–276. IEEE Press, 2014.

[62] 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.

[63] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Relaxed persist ordering using strand persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665, 2020.

[64] OSS-Fuzz. https://github.com/google/oss-fuzz.

[65] 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.

[66] 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.

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

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

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

[70] 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.

[71] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, page 273–287, USA, 2004. USENIX Association.