Vinter: Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems

Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa, *Karlsruhe Institute of Technology*

https://www.usenix.org/conference/atc22/presentation/werling

This paper is included in the Proceedings of the
2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the
2022 USENIX Annual Technical Conference
is sponsored by

VINTER: Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems

Samuel Kalbfleisch
Karlsruhe Institute of Technology

Lukas Werling
Karlsruhe Institute of Technology

Frank Bellosa
Karlsruhe Institute of Technology

Abstract

Non-volatile memory (NVM) is a new byte-addressable storage technology that is part of the processor's memory hierarchy. NVM is often exposed to applications via an in-kernel file system. To prevent data loss in the case of crashes, the file system implementation needs to be crash-consistent. Achieving crash consistency is difficult however, as special primitives need to be inserted at appropriate places in the program to ensure persistency in the presence of volatile caches.

We introduce VINTER, a new approach to automated NVM crash consistency testing designed for full systems, including unmodified kernel software such as file systems. By tracing NVM accesses of a full system via dynamic binary translation, we capture interactions between user and kernel space code. With such traces, our system efficiently generates relevant crash states using a heuristic that determines NVM locations significant for crash consistency. Finally, it extracts the semantic representation of each crash state. This makes the automatic detection of operation-spanning violations of crash consistency properties such as atomicity feasible. Our approach further aids in fixing detected bugs by representing how bugs originate from simulated crashes which are annotated by trace metadata.

Our evaluation on NVM file systems uncovers several previously unknown bugs, including bugs in the state-of-the-art file systems NOVA and NOVA-Fortis that lead to atomicity violations and data loss.

1 Introduction

Non-volatile memory (NVM) is a new storage technology that is byte-addressable and integrated in the processor's memory system. Applications can obtain a virtual memory mapping to access non-volatile memory pages directly with load and store instructions [40, 41, 52]. Such direct access enables persistency without a serialization step, but requires extensive reworking of the applications to ensure crash consistency: Modifications need to be flushed from volatile caches, and developers need to employ memory fences to enforce persistency ordering. Thus, programming for NVM has turned out to be difficult in practice [39].

As an alternative, unmodified applications can benefit from high-speed NVM by running on NVM file systems [5, 23, 26, 45, 48–51, 54, 56] which implement common user space interfaces such as POSIX [15]. Internally, these file systems store metadata and file data directly on NVM, which means that the same challenges for achieving crash consistency apply to these file systems as well.

Recent research has produced many approaches to detecting crash consistency bugs [8, 11, 27–30, 33]. However, we find that most of these approaches cannot easily be applied to file systems. Kernel software is often not supported at all or requires extensive code modification. In particular, static code analysis [11] and symbolic execution [33] are difficult to apply to full systems where a variety of user and kernel space code may interact. It is possible in theory to adapt a kernel file system to user space in order to apply a testing tool designed for user space software. However, this approach is likely to distort results, as the interaction between user and kernel space code can be a source of consistency bugs. For example, the NOVA file system [49] relies on memory barriers being performed when returning to user space.

We introduce VINTER, the virtualization-based NVM tester, a novel approach for testing crash consistency that supports full systems. Using binary translation, we trace relevant instructions such as load-store instructions or barriers in a virtual machine running unmodified kernel and user space code. From that trace, we generate *crash images* which represent possible NVM contents after a crash. We reduce the exponential search space by identifying NVM locations where the recovery code reads from the crash image. By extracting the *semantic state* of each crash image (e.g., a listing of all files in a file system), we can finally automatically verify crash consistency properties such as atomicity.

We use our prototype to test the NVM file systems NOVA [49], NOVA-Fortis [50] and PMFS [10] for crash consistency bugs. We find several new bugs in these file systems

ranging from atomicity violations to data loss and broken file system states. One specific bug we find in NOVA highlights the importance of testing unmodified software instead of higher-level methods such as manual code annotation: A generic Linux helper function for an uncached memory copy has an optimized assembly implementation for x86 that leaves unaligned data in the cache. NOVA uses this function to copy file data to NVM and is thus susceptible to data loss.

We identify the following major contributions of our work:

- Our solution traces NVM accesses using full system emulation with dynamic binary translation. It thus supports unmodified user and kernel space software.
- We apply heuristics to achieve efficient exploration of crash states, avoiding combinatorial state explosion.
- Through grouping simulated crashes by their semantic state, we introduce *automatic* testing of operation-spanning crash consistency properties such as atomicity.
- To help developers fix uncovered bugs, we introduce a representation of semantic crash states and their origins in simulated crashes which are annotated by trace metadata.
- Using our solution, we provide the first comprehensive analysis of NVM file systems for crash consistency.

2 Background and Related Work

In this section, we introduce the memory persistency model that VINTER builds upon and the crash consistency properties it can verify automatically. Then, we discuss related work.

2.1 Memory Persistency Models

Applications that access NVM need to pay close attention to the *memory persistency model* which codifies the architecture's guarantees about persistency. Multiple models have been proposed [5, 12, 18, 35]. We implement VINTER for the x86 architecture [17, 37] whose persistency model is based on *persistency epochs* [20]. The epoch model divides thread execution into persist epochs and guarantees that stores between epochs are strictly ordered, but not within the same epoch [5, 35]. Stores are buffered in x86 and need to be flushed from volatile caches with instructions such as `clwb` to be persisted [22, 37]. Alternatively, *non-temporal stores* bypass the caches. Memory barriers in the form of `fence` instructions provide *ordering points* that divide the epochs. In the following, we refer to all instructions that are relevant for the persistency model as *persistency primitives*.

2.2 NVM Crash Consistency

Crash consistency as a property of stateful applications is often only informally specified. Throughout this work, we use the following definitions and assumptions about the tested applications.

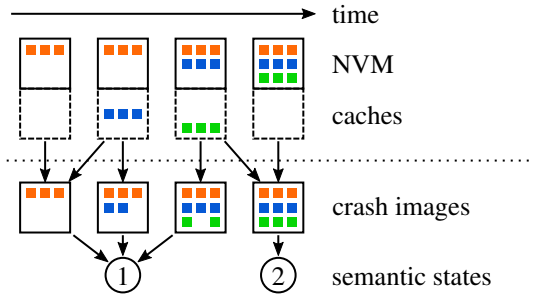

Figure 1: An atomic operation: All crash images (partially shown) recover to one of two semantic states.

We assume that the application keeps its persistent state in NVM. Depending on the access method, modifications may go through a volatile cache or may be reordered. After a crash, these modifications may be partially lost. Consequently, the NVM then contains all fully-persisted data (e.g., through cache flush instructions and memory fences) with a subset of the in-flight modifications applied. We call this a *crash image*.

The period during which a crash may occur is called the *pre-failure* execution [29]. When restarting the application from a crash image, its *post-failure recovery* code will read the data to recover its semantic state. By comparing all possible recovered states that can arise from crashes, we can describe the following crash consistency properties. An operation with all-or-nothing semantics fulfills *atomicity*. More formally, all crash images possible during an atomic operation result in one of two states: either the initial or the final state. Figure 1 illustrates such an atomic operation. From left to right, an application is modifying NVM contents step-by-step. As long as the modifications are not flushed from the caches, crash images with partial cache contents are possible. However, all but the final crash image recover to an identical semantic state. We can observe patterns like this when techniques such as journaling are in use. The final writes (green) mark the previous modifications as valid.

For non-atomic operations, in Section 3.4.1 we define a second, weaker property *Single Final State*: at the end of an operation, all possible crash images recover to the same semantic state. Either of these properties may be violated by different kinds of bugs. For example, a missing memory fence at the end of an operation may violate Single Final State: the CPU could reorder cache flushes, which results in crash images that miss some modifications.

Multiple methods exist that help applications to achieve crash consistency, including logging [46, 47], log structuring [4, 14, 24, 49, 55], and shadow paging [31, 34].

2.3 NVM Crash Consistency Testing

Since making NVM programs crash-consistent is difficult for developers, multiple testing approaches have been proposed

with varying degrees of automation. Many of the previous approaches rely on user-provided code annotations [29, 30], do not automatically confirm bug candidates [8, 29, 30, 33], and are limited to user space applications [11, 29, 33] or would need additional effort like kernel modification [8, 28, 30]. VINTER is free from these limitations.

Lantz et al.'s Yat [27] traces its target in a hardware-assisted hypervisor, constructs all possible crash images by brute force, and tests these by running an integrity checker such as `fsck`. It is able to test kernel space applications, but it relies on code modifications to trap persistency primitives due to limits of hardware-assisted virtualization. Due to its exhaustive crash image exploration, some test cases would take years to complete. Furthermore, Yat can only detect inconsistencies discovered by an integrity checker, and is not able to consider consistency across entire operations.

Liu et al.'s PMTest [30] requires that developers annotate their source code with persistency assertions. Their correctness is then evaluated at runtime. The obvious drawback is that the approach entirely depends on the quantity and quality of annotations which need considerable effort by developers.

XFDetector by Liu et al. [29] applies heuristics to detect a typical bug pattern. During runtime, it looks for read operations during post-failure recovery whose corresponding stores from pre-failure execution have not been explicitly persisted. To avoid false positive bugs, they require manual code annotations. Still, the bug candidates need manual screening.

Neal et al.'s Agamotto [33] applies symbolic execution to NVM user space programs. This allows arguing over many possible execution paths at once and can work with symbolic NVM. It only detects some bug patterns like unpersisted NVM locations after program termination. Thus, false positives can occur and manual vetting is required.

WITCHER by Fu et al. [11] focuses on testing user space key-value stores. The authors propose a testing pipeline that automatically confirms bugs. It uses heuristics based on dynamic and static analysis to choose interesting crash states and thus avoids exhaustive search. Use of static analysis and need for recompilation makes it difficult to apply to full systems, so we propose a new crash image generation heuristic solely based on dynamic analysis. WITCHER automatically detects violations of the *atomicity* of operations [18] by introducing "output equivalence checking" that compares with oracle states obtained before and after an operation. Our work extends upon this by also testing for a relaxation of atomicity. Furthermore, VINTER outputs a representation of all encountered semantic crash states for easy manual sighting when strong properties such as atomicity do not apply. VINTER also passes metadata through its testing pipeline in order to facilitate debugging and root cause analysis of uncovered bugs.

PMFuzz by Liu et al. [28] uses fuzzing for test case generation and is orthogonal to our work. PMDebugger by Di et al. [8] focuses on efficiently processing memory traces. Formal verification of NVM programs has been proposed [7, 13, 18].

2.4 File System Crash Consistency

The need for file systems to tolerate crashes is not new. Numerous works have proposed methods for checking crash consistency properties in file systems [19, 21, 25, 32, 36, 53]. Most of these approaches rely on instrumentation at the block layer and thus cannot be applied to NVM file systems. However, we see opportunities to adopt specific techniques to an NVM context. CrashMonkey [32] automatically generates test cases for a crash consistency checker. Our evaluation relies on manually written tests, but could be extended with a similar technique. Jaffer et al. [19] evaluate how file systems react to media errors common on solid state drives. Our approach could be extended in this direction, however, there is currently limited information on specific NVM media errors.

File system semantics. A common issue with file system crash consistency checking is that consistency semantics vary from file system to file system and are often only loosely specified. The POSIX standard [15], whose API most Unix file systems implement, does not define crash consistency semantics at all. Bornholt et al. [3] improve on this situation by creating crash consistency models for a few commonly used file systems. Rebello et al. [38] specifically look at error handling by file systems and applications for the `fsync` system call. These issues also apply to NVM file systems. For our analysis, NOVA [49], NOVA-Fortis [50] and PMFS [10] give strong atomicity guarantees for all metadata and file data operations, so complex models were not required.

3 Approach

We introduce a novel crash consistency testing approach to automatically finding bugs that violate crash consistency properties in unmodified NVM applications, including kernel space software. The key requirements for our solution are:

- To support kernel software such as file systems, it should work with *full systems* and should not be limited to user space software.
- It should not require manual development effort. In particular, no code annotations should be needed. Our solution even works with *unmodified* kernel and user space binaries without needing source code access.
- Reported crash consistency bugs should be *automatically confirmed*, and not be based on heuristics that allow for false positives.
- For good performance, it should *avoid exhaustive search* over all possible crash states, and only consider crash states that are likely to exhibit crash consistency bugs.
- It should not only look for single crash states that are obviously broken, but also consider *semantic, operation-spanning crash consistency* such as atomicity. For ex-

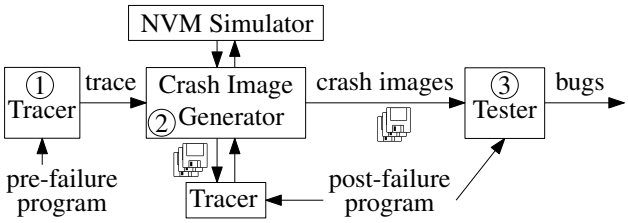

Figure 2: Overview of the testing pipeline. Components are in boxes; arrows are labeled with inputs or produced artifacts.

ample, this allows to determine whether a file system operation is atomic or has invalid intermediate crash states.

3.1 Overview

VINTER dynamically analyzes the execution of a system that interacts with NVM. The goal is to simulate crashes that may occur during a recorded pre-failure execution and find inconsistent states. The proposed framework consists of a testing pipeline made up of multiple components which we depict in Figure 2.

The pipeline's initial input is an operating system image, which includes all kernel and user space binaries, as well as a sequence of operations that specifies the program's pre-failure execution during which crashes should be simulated. Each step of the pipeline outputs an artifact that is fed to the next stage of the pipeline. In the final step, uncovered crash consistency bugs and a representation of encountered semantic crash states are output. The entire pipeline may be run multiple times with different test cases. Achieving comprehensive results depends on the quality and quantity of these cases being run in the system. Finding these is orthogonal to our work (e.g., PMFuzz [28]).

VINTER's *Tracer* makes use of full system emulation with dynamic binary translation to trace NVM operations performed by a system (§3.2). The resulting trace is fed into our *Crash Image Generator*. To be able to reconstruct arbitrary crash states, the Crash Image Generator uses an *NVM simulation* of a given memory persistency model to replay the trace on it (§3.3). At each ordering point (e.g., memory barrier), crash images are chosen by a heuristic that only considers in-flight stores at memory locations likely to be read by the recovery *(post-failure)* code, which avoids exhaustive exploration of crash states. The latter are determined by running the post-failure code on special crash images, hence the Crash Image Generator uses its own Tracer instance.

The *Tester* component finally takes crash images and extracts their *semantic (crash) states* (§3.4). Each of the pre-failure operations is then analyzed for violations of crash consistency properties such as atomicity. For example, if during an operation three different semantic crash states are observed, the operation has been shown not to be atomic. Uncovered bugs are finally output to the user of the framework, together with a representation of encountered semantic crash states and how they originate from the simulated crashes (§3.5).

The workflow of our testing pipeline is similar to the one in the WITCHER framework proposed by Fu et al. [11]. However, our crash image generation algorithm additionally makes use of the Tracer as part of our heuristic for choosing crash images (we further compare our solution in §2.3).

3.2 Tracer

The Tracer is a full system emulator and is used in the testing pipeline for the following purposes:

- During pre-failure execution, we trace *writes* to NVM as well as invocation of *persistency primitives*, as this allows reconstruction of arbitrary NVM crash states (§3.3).
- Further, we trace *reads* to NVM during post-failure executions as part of our crash image generation heuristic described in Section 3.3.
- Optionally, the Tracer may be used by the Tester component for running the post-failure recovery on the generated crash images to extract their semantic states (§3.4). Tracing NVM accesses may be disabled, but hypercalls can be used (see below).

In common architectures, NVM is accessed directly via load and store instructions to virtual memory that is backed by NVM. Further, persistency primitives (such as cache line flushes and memory fences) are usually implemented as dedicated instructions. To intercept these events, we base the Tracer's processor emulation on *dynamic binary translation* [43]. This allows full control over the emulated system, and is transparent to the tested program stack, thus binaries may remain unmodified. In contrast, hardware-assisted virtualization (as used by Yat [27]) is faster, but does usually not provide hooks for all events that are of interest (Yat requires recompilation).

During the translation of basic blocks from the emulated architecture to the host architecture, the Tracer adds instrumentation code for persistency primitives to the translated code. Further, the memory abstraction layer of the emulator intercepts memory writes and reads to address ranges backed by simulated NVM. The intercepted events are then output in a *trace log* with associated data. Data includes instruction operands (e.g., cache flush addresses) or memory contents updated by a store to NVM.

Hypercalls. We allow the emulated system to signal the following events as hypercalls to the testing framework, which are additionally recorded in the trace log:

- *Checkpoint* hypercalls during pre-failure execution separate semantic operations. We remind that input to the testing framework is a sequence of operations that will each be separately tested for operation-spanning crash consistency. In practice, this may be in form of a sim-

ple user space executable that calls the tested program's operations and emits hypercalls in between.
- *Success indication* hypercalls may occur during post-failure recovery. If none is emitted, it is assumed that recovery from the specific crash image has failed.

The use of hypercalls is not mandatory for the approach, but it improves the testing workflow for example by allowing separation of operations during pre-failure execution.

Metadata. During pre-failure execution, we log metadata about traced events (such as NVM stores), particularly their location in form of a call stack as traced by the emulator. Although not required to detect crash consistency bugs, it helps a developer to investigate uncovered crash consistency bugs. We retain this metadata when generating crash images later in the pipeline as a way to understand how a crash needs to occur to lead to a certain invalid semantic crash state. Further, additional events helpful for debugging may be traced and logged, such as system calls.

3.3 Crash Image Generator

The Crash Image Generator component takes a pre-failure trace as its input, and it outputs crash images to be passed into the Tester (§3.4). A crash image is a string that describes the binary contents of the NVM in a single crash state that can occur according to the memory persistency model at an arbitrary point of program execution.

Motivation and challenges. In memory persistency models that are based on persistency epochs [5, 35] (including x86 [20]), the set of all possible crash images can be constructed as follows. At each ordering point (usually memory fences), apply any possible subset of potentially unpersisted in-flight stores on top of the memory contents that are already guaranteed to be persisted [8, 11, 27–30, 33]. This works because stores can only become irreversably persistent at ordering points. Not all subsets may be allowed and order of stores can be relevant depending on the specific memory model [35, 37, 42], such as x86 with intra-cache-line ordering.

Although considering any possible crash state would be comprehensive, it is impractical. Yat by Lantz et al. [27] does so, but the authors find that some test cases would take several years to complete due to exponential explosion in number of crash images. Thus, many other approaches do not actually generate and test crash images, but only apply heuristics on the observed program execution to detect typical bug patterns [8, 29, 30, 33]—this either allows false positives to occur or requires extensive manual code annotations.

Solving this problem requires reducing the number of tested crash images. The chosen subset of crash images should ideally not hide bugs. Accordingly, the chosen crash images should have some properties that makes them relatively likely to exhibit bugs. The recent WITCHER framework proposed by Fu et al. [11] aims to solve this challenge for user space key-value stores: From a mix of static and dynamic analysis, they infer *likely invariants* regarding the persistency of program data that have presumably been intended by the programmer. Then, they choose crash images that violate these invariants and test if these indeed violate crash consistency. This approach works well for key-value stores, but particularly the reliance on static analysis makes it difficult to apply to full systems, where a variety of user and kernel space code may interact.

Proposed crash image generation heuristic. In contrast, our heuristic only relies on dynamic analysis. For determining crash consistency, only the semantic state as recovered by the application in post-failure recovery is relevant (§3.4). Conversely, the semantic crash state can only have been influenced by *memory locations from which the post-failure execution has read.* Our crash image generation heuristic is based on this idea. When choosing subsets of in-flight stores during crash image generation, our heuristic only considers stores likely to be read by the post-failure recovery. Other stores may be ignored.

We observe that techniques such as journaling and log structuring, both common in file systems, cause the following access pattern: The program writes a journal entry, flushes it completely to NVM and only marks it valid after a store fence. The journal entry may have an arbitrary size, resulting in a large number of in-flight stores. However, considering subsets of these stores for crash image generation is not useful. After a failure, the recovery would only read a journal entry that is marked valid.

According to this observation, we base the decision on whether a store is likely to be read in post-failure recovery on the following assumption: If an NVM location is not read during the post-failure stage on the image where all unpersisted stores are applied, the post-failure stage will likely also not read this location when an arbitrary subset of in-flight stores is applied. Therefore, the heuristic limits the generated crash images to variations of stores that *are* likely to be read during the post-failure stage's execution. As with any heuristic, the assumption may not always hold. We evaluate its effectiveness in Section 6.2.

For the heuristic to capture all relevant NVM locations, the post-failure recovery should read all relevant state, for example by running code that serializes all state (as in the state extractor to be introduced in Section 3.4).

The heuristic's underlying idea of observing interactions between pre- and post-failure executions is based on Liu et al.'s XFDetector framework [29]. However, XFDetector uses these observations directly to detect bug patterns, but does not automatically confirm bug candidates. It further requires developers to manually annotate their code to mark memory belonging to commit variables; it is further not directly compatible with checksumming mechanisms as used by file systems.

NVM simulation. To be able to replay the trace and reconstruct the possible NVM crash states, we assume a simu-

lator of the architecture's memory persistency model. It holds the *guaranteed persisted memory* content as a binary string, and further a list of *in-flight stores* that have not yet been explicitly persisted. Each of these stores further retains associated metadata from the trace. It further needs to provide functions for applying stores and persistency primitives (e.g., cache line flushes and ordering points). An efficient data structure and algorithm for processing in-flight stores have been proposed by Di et al. [8].

Resulting algorithm. The algorithm processes the pre-failure trace from beginning to end and replays each NVM write operation and persistency primitive invocation on the NVM simulation. At each ordering point, it generates crash images in the following way:

1. Obtain a copy of the current NVM's guaranteed persisted memory, and apply all stores on it. We obtain NVM_{full}.
2. Instantiate the Tracer with NVM_{full} as initial NVM contents. Execute and trace the post-failure recovery.
3. Look for "read" operations in that post-failure trace to NVM addresses that have overlapping in-flight stores.
4. Consider all subsets of these cross-failure read in-flight stores, and apply each subset to the guaranteed persisted memory and emit the resulting crash images.
5. Continue with replaying the pre-failure trace.

As an optimization, we ignore ordering points if no stores to NVM have occurred before the last one. Further, if there would be too many subsets of cross-failure read in-flight stores according to a configurable threshold, we choose a random selection of these subsets.

Metadata. It is possible for the same crash image (merely a binary string) to be emitted multiple times, but at different ordering points and with different subsets of stores applied. We deduplicate crash images and attach metadata to each image that describes all its *origins*. Each origin includes the (un)persisted in-flight stores' metadata (containing the stack trace that led to a store), the last ordering point's ID, and the last checkpoint's ID. Crash images are further grouped by checkpoint (i.e., the semantic operation they occur during).

3.4 Tester

In the previous step of VINTER's testing pipeline, we have generated crash images that simulate crashes at multiple points during the traced pre-failure program execution. In this final step, the Tester component analyzes each operation's crash images for the occurrence of crash consistency bugs. We begin by defining crash consistency properties (§3.4.1) and then describe how the Tester component discovers violations of the properties from a set of crash images (§3.4.2). Our testing approach is an extension of Fu et al.'s "output equivalence checking" [11].

3.4.1 Crash Consistency Definitions

The central idea is that an NVM image can be mapped to an application-specific well-defined *semantic state* that describes the intended meaning of the persisted data. The same semantic state can possibly be encoded by different NVM images. We use $\mathcal{S}$ to describe the set of semantic states, and $\perp$ ($\perp \notin \mathcal{S}$) to denote that an NVM image is not recoverable from because it is faulty. The *state extractor* function E maps NVM images to semantic states, for which we use the notation $E : \{0,1\}^* \to \mathcal{S} \cup \{\perp\}$.

We recall that we allow traces to be separated by *checkpoints* $(c_1, c_2, \dots)$ that are signaled by the running program through hypercalls (§3.2). We define *checkpoint intervals* $[c_i, c_{i+1}]$—called *operations* in the following—that each induce a subsequence of a trace that contains all its recorded events between checkpoints c_i and c_{i+1}. We use checkpoints to separate different semantic operations during the pre-failure execution traced in the beginning of our testing pipeline. We define two crash consistency properties which, depending on the operation, may be considered a requirement for crash consistency of the operation.

We propose the following new property:

Definition 1 (Single Final State, SFS). A *checkpoint* c_k is single-final-state crash-consistent ($\text{SFS}(c_k)$) if and only if all crash images $N_k \subset \{0,1\}^*$ that can result from crashes in the trace exactly at checkpoint c_k result in the same state $\neq \perp$, or formally: $\exists s \in \mathcal{S} : \{E(n) \mid n \in N_k\} = \{s\}$.

For $s \in \mathcal{S}$, we write $\text{SFS}(c_k, s)$ if c_k is single-final-state crash-consistent and $\{E(n) \mid n \in N_k\} = \{s\}$. Further, an *operation* $[c_i, c_{i+1}]$ is single-final-state crash-consistent if and only if c_{i+1} is single-final-state crash-consistent.

SFS is a property that is useful to require even when an operation is not considered atomic; in that case, intermediate states are allowed, but as soon as an operation returns, a crash may not yield any intermediate states anymore.

We further define the well-known atomicity property in our context:

Definition 2 (Atomicity). An operation $[c_i, c_{i+1}]$ is atomic if and only if c_i and c_{i+1} are both single-final-state crash-consistent, and all crash images $N_{[i,i+1]} \subset \{0,1\}^*$ that can result from crashes anywhere between checkpoints c_i and c_{i+1} result in either of two states $\neq \perp$, or formally:

$$\exists s_{\text{before}}, s_{\text{after}} \in \mathcal{S} : \text{SFS}(c_i, s_{\text{before}}) \wedge \text{SFS}(c_{i+1}, s_{\text{after}}) \\ \wedge \{E(n) \mid n \in N_{[i,i+1]}\} = \{s_{\text{before}}, s_{\text{after}}\}.$$

Atomicity means that operations execute, from the point of view after crash recovery, in an all-or-nothing fashion: Either an operation is fully run (s_{after}) or not at all (s_{before}). No intermediate states should occur and all states should be recoverable. This means that no more than two states may be observable after recovering from arbitrary crashes between

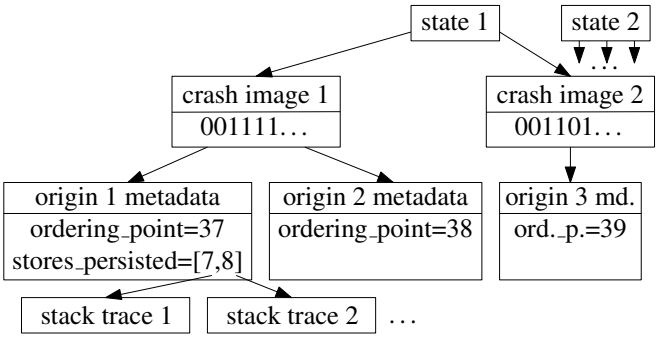

Figure 3: Representation of an operation's semantic states and their origins' metadata as output by VINTER.

the two checkpoints of an operation. An atomic operation is obviously also SFS crash-consistent, but SFS crash-consistent operations exist that are not atomic.

The state extractor E will usually be implemented by running the application's post-failure recovery on the given crash image including a step that serializes the state. Additional tests may be run after the recovery to ensure correctness (such as `fsck` on file systems). In this case, if a hypercall indicating success is not issued by the post-failure recovery, the state is mapped to $\perp$. The Tracer component can be reused to provide a virtual environment that implements these hypercalls.

3.4.2 Testing Crash Consistency Properties

VINTER's Tester component finds crash consistency violations given a set of crash images from the previous step of the testing pipeline. As its input, we assume a map of (already deduplicated) crash images to lists of the metadata that describe an image's (possibly multiple) origins (§3.3).

The Tester builds a table $M_{\text{cp}\mapsto\text{states}}$ that maps checkpoint IDs to semantic states which originated from crash images where the most recent checkpoint hypercall had the corresponding ID. As an image can have multiple origins with different checkpoint IDs, a semantic state may be mapped to more than one checkpoint. It also builds another similar table $M_{\text{cp}\mapsto\text{fin. states}}$, with the exception that only images are considered that result from crashing directly at a checkpoint boundary, thus only considering the "final" states at the end of an operation.

It is easy to find violations of crash consistency properties by scanning the tables:

- The *Single Final State* property of a checkpoint is valid if and only if $M_{\text{cp}\mapsto\text{fin. states}}$ contains exactly one state for that checkpoint.
- The *Atomicity* property of an operation $[c_i, c_{i+1}]$ is valid if and only if c_i and c_{i+1} are SFS crash-consistent (with $M_{\text{cp}\mapsto\text{fin. states}}[c_i] = \{s_{\text{before}}\}$ and $M_{\text{cp}\mapsto\text{fin. states}}[c_{i+1}] = \{s_{\text{after}}\}$) and $M_{\text{cp}\mapsto\text{states}}[c_i] = \{s_{\text{before}}, s_{\text{after}}\}$.

Violations of these properties are reported by the Tester. Further, for each operation it outputs a representation of all encountered semantic crash states with their associated crash images and the crash images' origins (Figure 3). This representation helps with root cause identification as we cover in the following section.

3.5 Bug Analysis

The Tester's output lets developers determine whether crash consistency bugs have been uncovered. Trivially, if the extracted semantic states include an unrecoverable state ($\perp$), a bug has been uncovered. Otherwise, in the simple case, a developer specifies that a certain operation should hold one of the previously covered crash consistency properties, and then a violation reported by the Tester implies a bug. However, the crash consistency semantics of an operation may be more complex or even unclear and may allow multiple intermediate states. The developer can then manually inspect the encountered semantic states for their validity (as output in the maps $M_{\text{cp}\mapsto\text{states}}$ and $M_{\text{cp}\mapsto\text{fin. states}}$). Alternatively, they can automate the decision by implementing an operation-specific predicate that validates whether for a set of semantic states the operation is crash-consistent.

Once bugs have been discovered, VINTER aids developers in understanding their root causes by representing how crash-consistency-violating semantic states originate from simulated crashes. As depicted in Figure 3, every semantic state keeps a link back to the crash images it was extracted from, and each crash image keeps a link to the simulated crashes' states (origins) it was created from. As a crash image only represents the NVM's binary contents, multiple different crash images may lead to the same semantic state (e.g., an uncommitted journal entry does not contribute to the semantic state). Further, crashing at different ordering points or with different subsets of in-flight stores applied, as stored in the "origin metadata," may lead to the same crash image.

Each crash image origin includes a list of in-flight stores at the ordering point (fence) where the crash was simulated, as well as the subset of in-flight stores applied to obtain the crash image. Each of the ordering points and their in-flight stores is annotated with its stack trace (as logged by the Tracer) and thus maps back to its source code location.

By inspecting the origins of a crash-consistency-violating semantic state, developers can understand where and how a crash needs to occur to trigger the bug (as in source code location and persistency status of NVM data). This eases root cause identification. For example, if *all* of an invalid state's origins have a *strict subset* of in-flight stores applied (i.e., the crash image at the same ordering point with all in-flight stores applied does not lead to the invalid state), then introducing more persistency ordering constraints (fences) may likely fix the bug.

We present two concrete bug analyses in Section 5.3.

4 Implementation

We implement a prototype of VINTER for the 64-bit x86 architecture [17]. Its tracer is based on PANDA [9] and PyPANDA [6], which provide a platform for dynamic analysis built on QEMU [1]. QEMU is a full system emulator based on dynamic binary translation. Core parts of our prototype are written in Python. Therefore, while its performance is sufficient for our file system evaluation, considerable improvements are likely possible without changing the general approach. VINTER's prototype is available and described in Appendix A.

Our NVM simulator implements x86's memory persistency model [17, 37]. By dividing the simulated NVM into segments of 64 bytes and keeping ordering of in-flight stores within these lines, VINTER respects intra-cache-line ordering constraints. This guarantees ordering even of non-temporal stores, which is a higher guarantee than the architecture gives. Setting the segment size to 8 bytes would allow testing reordering of non-temporal stores, but would also break intra-cache-line ordering which some NVM file systems rely on [10].

5 File System Crash Consistency

The crash consistency testing tool we have described so far does not have any parts specific to file systems and could be applied to arbitrary software running in a virtual machine. In this section, we describe how to apply it to NVM file systems. Then, we present the results of our analysis of NOVA [49], NOVA-Fortis [50] and PMFS [10].

Our testing pipeline requires an application-specific *state extraction procedure* for mapping from a crash image to its semantic state. The Tester component boots a virtual machine with the crash image, runs the procedure, and records the output or a failure state $\perp$ in case of errors. We implement file system state extraction as follows:

1. Mount the file system read-only.
2. Traverse the file system and output a serialized representation of each file.

We need to mount the file system read-only to prevent inadvertent changes to metadata such as file access timestamps. The file systems we target adhere to the POSIX standard [15], which allows us to build a generic state extractor for all POSIX-compatible file systems. For each file (including regular files, directories, symbolic links) we output a serialization of its path, its contents, its type, and most metadata from the *stat* structure[1].

After the state extraction completes, we verify that the file system can still be modified. We remount the file system as writable and run an additional test-case-specific command that modifies some of the files or directories used in the test. If any of these operations fail, we mark the state described by this crash image as failure state $\perp$.

Our notion of consistency only encompasses the file system state as visible via the file system API. The underlying assumption is that if the file system is still properly readable and writable, its state can be considered consistent without the need to inspect its internal state.

For testing more traditional file systems that depend on separate integrity checkers (`fsck`), these could also be run as part of the state extraction procedure and map to the failure state $\perp$ if the integrity checker fails.

5.1 File System Setup

For each tested file system, we need a corresponding virtual machine image. To reduce the time required for tracing, we hand-craft minimal VM images consisting of a statically-linked Linux kernel image with a user space based on BusyBox [44]. We do not use an init system. Instead, we let the system spawn a shell that accepts test commands over a virtual serial console. Consequently, there are no unrelated processes running in the background during tracing.

5.2 Test Cases and Results

We manually craft 16 test cases consisting of operation sequences which cover most basic file system operations. Figure 4 shows a summary of the test cases and the results. Most test cases correspond to basic file system operations given in `monospace` font. The "atime" and "[cm]time" test cases update the corresponding timestamps as side effects of a file read or directory operations[2], whereas "touch" uses a system call for that purpose. We test three variants of the `rename` operation: a rename that overwrites an existing file (overwrite), moving a directory into another (directory), and changing the file name to a longer one (long name). The "long name" test case creates a file with a long file name and writes to it. Test cases with "long" file names use a name that exceeds the cache line size (which is 64 bytes). Finally, "update" modifies a small part in the middle of a larger file.

In total, VINTER finds previously unreported bugs in 7 out of 16 test cases for NOVA. We analyze these bugs manually and find three root causes. First, we observe missing cache flushes when NOVA writes unaligned data to NVM which lead to data loss. This issue manifests in our test cases "write" and "symlink," but could also occur in "append" and "update" (marked with an asterisk) depending on the length of the written data. The test cases with long filenames (i.e., longer than cache line size) suffer from the same issue and result in files where reading the metadata with `stat` fails. Second, the rename operation is not atomic. We observe crash states where the renamed file or directory is completely missing.

[1] We exclude runtime properties such as device IDs and preferred I/O block size.

[2] Adding or removing files from a directory updates the directory's change and modification timestamps.

	write	append	atime	[cm]time	chmod	chown	link	symlink
NOVA	💔🍎	✓*	✓	✓	✓	✓	⚛	💔🍎
NOVA-Fortis	👓	💔	✓	👓	✓	✓	👓	💔🍎
PMFS	✓	✓	✓	(⚛)	✓	✓	✓	✓

	mkdir rmdir	rename overwrite	rename directory	rename long name	touch	long name	unlink	update
NOVA	✓	⚛💔	⚛💔	⚛💔👓	✓	👓	✓	✓*
NOVA-Fortis	👓	⚛💔👓	⚛💔👓	⚛💔👓	👓	👓	👓	💔
PMFS	(⚛)💣	(⚛)💣	✓	✓	✓	(⚛)	(⚛)💣	✓

💔 data loss 💣 crash ⚛ atomicity violation 👓 read/write fails after recovery 🍎 multiple final states (SFS violation)

Figure 4: Crash consistency bugs discovered by VINTER.

In the following section, we give a detailed analysis of these two bugs. Third, creating hard links is not completely atomic. Our tool detects a crash state where the original file's link count (`st_nlink`) is incremented, but the new link does not yet exist.

As NOVA-Fortis is an extension of NOVA with the addition of checksumming and parity, it shares most of the issues we find in NOVA. However, we see three additional failing test cases. Our tool discovers intermediate crash states where both data and the checksum over that data are only partially persisted. This leads to checksum errors during recovery and errors when trying to read or write the affected files. We also observe an instance where the additional data integrity mechanisms help: In the "write" test, NOVA-Fortis does not suffer from data loss since it can recover the unpersisted data from parity. However, it appears that NOVA-Fortis does not protect all data that way: The data loss in the "symlink" test case shares the same root cause as for "write," but still occurs in NOVA-Fortis.

PMFS suffers from fewer failing test cases than the NOVA variants. We observe a minor atomicity violation in test cases that remove or overwrite a file. Before the file disappears, crash states exist where the file has updated change and modification timestamps. A more serious issue can occur when removing files in the root directory: Crash states are possible where mounting the file system results in a failing assertion, which leads to a crash of the PMFS kernel module.

We reported all NOVA and NOVA-Fortis bugs to the developers[3]. We did not report PMFS issues since it is not maintained anymore.

5.3 Analysis

VINTER has discovered several previously unreported crash consistency bugs in the NOVA variants. In the following, we exemplarily provide a detailed analysis of two of these bugs that highlights advantages of our crash consistency testing approach. Furthermore, the analysis illustrates how VINTER helps manual analysis with the metadata carried throughout the testing pipeline and presented as part of the Tester's report.

[3]Issue IDs 105, 116, 121–125; each accessible at https://github.com/NVSL/linux-nova/issues/<ID> (also on archive.org)

unpersisted stores	call stack of store (metadata)
0x2db008 ↦ `l`	`__copy_user_nocache`
0x2db009 ↦ `d`	`do_nova_inplace_file_write`
0x2db00a ↦ ↲	...
	`vfs_write`
	...

Figure 5: The unpersisted stores after writing `HelloWorld`↲ to a file in NOVA. The bytes `l`, `d`, ↲ are the last three bytes of the string.

event	operand	instr. (metadata)
syscall	sys_write(fd=1, buf='HelloWorld\n', *n*=11)	
...	...	...
write (NT)	0x...0 ↦ `HelloWor`	movnti qword [rdi], r8
write (T)	0x...8 ↦ `l`	mov byte [rdi], al
write (T)	0x...9 ↦ `d`	mov byte [rdi], al
write (T)	0x...a ↦ ↲	mov byte [rdi], al
fence		sfence (store fence)

Figure 6: Excerpt of the trace that shows how the file contents are written to NVM. No flush operations follow on the cache line belonging to the temporal stores.

5.3.1 Incompletely Persisted Data

We first analyze the data loss bug occurring in the test case "write" (see Figure 4). The test case creates a file and writes the string `HelloWorld`↲, as in the shell command `echo HelloWorld > /mnt/myfile`. Our tool detects a violation of Single Final State and records partially persisted file contents where up to three bytes at the end are replaced with zero (e.g., `HelloWor`0 0 0 and `HelloWorl`0 0).

First, we take a look at the unpersisted stores as reported

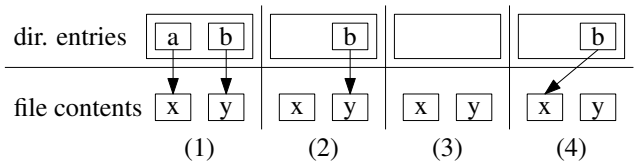

Figure 7: Observed crash states in NOVA during a rename of file *a* to *b* in a directory, intending to replace *b*. Boxes represent files. (1) is the initial state.

by the Tester component. As pictured in Figure 5, each store is associated with a call stack. We see that the stores originate from a `write` system call and that NOVA uses the function `__copy_user_nocache` to write to NVM.

Next, we inspect the NVM trace pictured in Figure 6. It captures the system call entry as well as all writes to NVM, cache flushes, and memory fences. Non-temporal writes (NT) are distinguished from temporal writes (T) as the former bypass the volatile caches. We see that the helper function `__copy_user_nocache` writes the first eight bytes of the string to NVM with a single non-temporal store, then writes the remaining bytes one by one with temporal stores. As there is no cache flush operation, these bytes end up unpersisted.

Further investigation in the Linux source code shows that `__copy_user_nocache` has an architecture-specific implementation for x86 implemented in assembly. As the function was written with performance and not persistency in mind, the Linux developers deemed it acceptable to use temporal stores for unaligned data.

This bug highlights the importance of testing *unmodified* software. Using an approach that relies on source code annotations to trace NVM events, a developer would likely assume that the Linux helper function works correctly and would annotate it accordingly without following through to the architecture-specific assembly implementation.

5.3.2 Data Loss During Rename

Second, we analyze the atomicity issues of the `rename` operation in NOVA. As depicted in Figure 4, our tool detects atomicity violations and data loss for all variants we test (overwriting a file, renaming a directory, and a normal rename with long file names). We visualize the crash states for the overwrite variant in Figure 7. In addition to the initial state (1) and the desired final state (4), we observe a state where the file to be renamed is missing (2), as well as a state where the target file is also missing (3). For the other two test cases that do not overwrite files, we observe crash states similar to (2) where the file or directory is missing.

We find that both invalid crash states from Figure 7 have at least one crash image associated with an "origin" where all in-flight stores were persisted (i.e., includes all volatile cache lines; §3.5). Consequently, the bug does not stem from mistakes with NVM persistency primitives. We inspect the call stacks for these crash images and find that they correspond to different locations in the `nova_rename` function. By inserting early `return` statements at these locations, we can reproduce the issue without our testing framework, thus confirming the issue.

This bug shows that our testing pipeline is also capable of finding persistency issues that are not specific to NVM. By carrying metadata such as stack traces through the pipeline, our framework makes bug verification easy.

6 Evaluation

In the previous section, we have shown that our approach is capable of finding new crash consistency bugs in file systems. We now give a more complete picture of our prototype's performance by answering the following questions: Is the approach comprehensive or does it miss certain types of known crash consistency bugs? How effective is our crash image generation heuristic at reducing the search space of possible crash images? How big is the slowdown incurred by tracing and how fast are the other stages of the testing pipeline?

6.1 Completeness

It is well known that—outside of formal methods—software testing can only prove the existence of bugs, but not their absence. Nevertheless, we would like to evaluate the *completeness* of our approach: Does it miss certain types of real-world crash consistency bugs?

To this end, we sight all 45 issues and patches available on the public NOVA bug tracker which were submitted between 2019-01-01 and 2021-10-30. We include issues and patches that are related to crash consistency bugs. We exclude #110[4] as according to the bug description, recovery from a crash only leads to error messages but no observed wrong behavior. We exclude #98, as according to the bug report, it requires an NVM capacity $> 128\,\text{GiB}$, which is infeasible with our evaluation setup. We are left with four different patches (#89, #92, #95, #109) that fix crash consistency bugs in NOVA and NOVA-Fortis.

To assess whether our prototype is able to find these bugs, we evaluate our prototype on a commit before and after each bug, and compare the Tester component's reports. This way, we can ignore the crash consistency bugs we describe in Section 5 which may already be present in older NOVA versions.

Our prototype is able to find all of the known bugs chosen by our evaluation methodology. There is no need for special test cases. All bugs are triggered by at least one of the test cases we describe in Section 5.2.

[4]The numbers are IDs of issues or patches in the NOVA bug tracker and can be accessed at URLs of the following form: https://github.com/NVSL/linux-nova/issues/<ID> (also on archive.org)

	$\mu \pm \sigma$ [s]
total elapsed Tracer process time	6.37 ± 0.11
↳ boot (only minimal instrumentation)	1.63 ± 0.03
↳ trace (from outside)	4.15 ± 0.10
↳ trace (in guest, portion of command)	3.02 ± 0.05
execution in guest with raw PANDA	0.09 ± 0.00

Figure 8: Runtimes measured while tracing the pre-failure command of the "write" test case in NOVA. 20 runs, μ is mean and σ is sample standard deviation.

	$\mu \pm \sigma$ [s]	
total elapsed process time	83.82 ± 0.53	
↳ boot	1.63 ± 0.03	
↳ Crash Image Generator	37.87 ± 0.33	
↳ cross-failure tracing (heuristic)	2.00 ± 0.28	$\times 12$
↳ Tester	43.65 ± 0.35	
↳ reset to snapshot & load image	0.08 ± 0.01	$\times 31$
↳ run dumper command (PANDA)	1.04 ± 0.12	$\times 31$

Figure 9: Runtimes of the Crash Image Generator & Tester process when processing the trace from Figure 8.

6.2 Effectiveness of Heuristic

We evaluate whether our proposed crash image generation heuristic based on cross-failure reads (§3.3) helps with efficiency as intended. To this end, we modify our prototype to consider *all* lines with unpersisted stores for crash image generation, rather than only lines overlapping with the unpersisted cross-failure reads as reported by our heuristic. We run this modified prototype on all NOVA test cases (§5.2).

We compare both the number of unique crash images generated (accumulated over all test cases), and the semantic crash states discovered by the modified prototype with that of the original prototype. 2 466 unique crash images are produced by the modified prototype without the heuristic, versus only 438 crash images by the original prototype[5]. Thus, the prototype without the heuristic needs to test approximately 5.6 times as many crash images. This results in an increased runtime, but it still discovers only the same semantic crash states as the original prototype using the heuristic.

We additionally analyze how the heuristic reduces the number of in-flight stores considered for crash image generation. In 47 out of 178 applications of the heuristic during all NOVA tests, the recovery code does not read any in-flight stores. We manually verify that these cases occur during journaling in NOVA by checking tracing metadata. In 94 applications of the heuristic, all cache lines with in-flight stores are read. In the remaining 37 heuristic applications, the cross-failure reads make up a strict subset of in-flight stores.

6.3 Performance

Even though performance was not a priority of our prototype, we evaluate its performance to show that the approach is sufficiently fast for testing file systems. We benchmark the Tracer's performance with the pre-failure command of the "write" test case (see §5.2) on the NOVA evaluation target on an Intel Xeon E5-2620 v4 CPU. We depict the results in Figure 8. Compared to the runtime in raw PANDA (i.e., binary translation without tracing), we observe a slowdown of approximately factor 34. The resulting traces each have $\approx 304\,438$ events and are each ≈ 11.73 MiB in size (in a simple textual format; compressed only ≈ 0.15 MiB).

As the Crash Image Generator and Tester run in the same process in our prototype, we show the execution time of both combined in Figure 9. 12 crash images at fences are used as input for the cross-failure heuristic. The tester processes 31 unique crash images that stem from 77 origins (§3.3) and result in seven unique semantic crash states. We find that the Crash Image Generator and Tester contribute roughly equally to the execution time.

With metadata tracing enabled, the runtime of the Tracer increases significantly to $78.70\,\text{s} \pm 0.75\,\text{s}$, whereas the runtime of the Crash Image Generator and Tester only increases slightly to $84.84\,\text{s} \pm 0.60\,\text{s}$. We argue that performance of metadata tracing mode is not very relevant in practice: Test cases can be tested without metadata tracing, and if a crash consistency bug is uncovered, the affected test cases can be rerun with metadata tracing enabled. Nevertheless, performance can presumably be significantly improved.

In total, all our test cases from Section 5.2 take approximately 24 minutes to execute sequentially on the NOVA evaluation target (without metadata tracing). This includes additional steps such as compressing traces. As test cases can be analyzed in parallel, the whole testing time can be reduced to only a few minutes.

7 Discussion

VINTER fares well at finding new bugs. Its testing pipeline is highly automated and the manual effort required for setup and interpretation of its results is low. In the context of file system testing, the Single Final State property has turned out to be useful, as even if operations are not considered atomic, this property should hold for most file system operations after a call to `sync`. Even in cases where the crash consistency semantics of a file system operation are not clear, VINTER's representation of semantic crash states makes manual vetting feasible. It has turned out to be useful to test modifying the file system after recovery as part of the state extraction procedure, as some of the bugs only become visible when such an operation fails.

[5] We have only performed a single run for each test case; the generated crash images in each run can slightly vary due to nondeterministic guest execution.

We find that severe crash consistency bugs can be found in NVM file systems by only testing primitive file system operations, with no need for complex interaction between multiple operations. This is contrary to the bugs discovered by CrashMonkey [32], which all appear to involve more complex operation sequences. We see two potential reasons: First, programming for NVM with its byte-level access and persistency semantics is much more complicated than the programming pattern for traditional file systems, where updated sectors are first built in DRAM and then transferred to block storage. Second, the tested file systems are of relatively young age, and are still research prototypes not aimed at production usage (and in case of PMFS, even unmaintained). Testing more complex operation sequences on our evaluation targets might yield even more bugs.

8 Conclusion

Crash consistency is difficult to achieve in non-volatile memory (NVM) software. Existing works on NVM crash consistency testing for kernel software are either inefficient or incomprehensive and not automated, and none consider crash consistency and atomicity among entire operations. This makes existing work unsuitable for comprehensive file system crash consistency testing.

In this work, we have introduced VINTER, a new approach to automatic testing of non-volatile memory software. VINTER consists of an automated testing pipeline that traces executions of full systems that use NVM, simulates crashes, and finally tests the resulting crash images for consistency. We use VINTER to find crash consistency bugs in NVM file systems, including the state-of-the-art file systems NOVA [49] and NOVA-Fortis [50]. Our evaluation uncovers several bugs in all tested file systems, many of them previously unreported. The bugs lead to issues such as data loss, kernel crashes, and unwritable files.

To summarize, our approach is general as it is compatible with differents kinds of software including kernel code, easy to apply as it is largely automated and does not need code modifications, and has been shown to find new bugs in existing software.

8.1 Future Work

We lay out possibilities for future work on our subject. We see several areas for improvement:

Support for persistent caches. Some recent processors ensure that all data in the CPU caches is written out to persistent memory in case of a power failure (e.g., eADR [16]). With persistent caches, the programmer no longer needs to use cache flushes to write out data to NVM. This greatly reduces the potential for crash consistency bugs, but does not entirely eliminate them. For example, the NOVA bug we describe in Section 5.3.2 would still occur on an eADR-enabled system. VINTER could support detecting crash consistency bugs on eADR systems by considering prefixes of all temporal writes instead of arbitrary subsets during crash image generation (§3.3). Choosing subsets of in-flight stores would however still be applicable to non-temporal writes since these are weakly-ordered.

Fault injection. Some NVM software including NOVA-Fortis [50] intends to be resilient against media errors. VINTER could be extended with fault injection to test robustness against corruption.

Heuristics. Bug detection or crash image generation heuristics that observe control or data flow such as those proposed by Fu et al. [11] could be adapted to our approach.

Evaluate NVM operating systems. VINTER could not only be applied to file systems, but also to entire operating systems targeted for NVM, like Bittman et al.'s Twizzler [2]. Twizzler removes the file system from the OS interface, and instead allows applications to directly allocate NVM.

File system test cases. Our file system evaluation could be extended by running automatically generated test cases in a large scale similar to Mohan and Martinez et al.'s CrashMonkey [32].

Traditional file systems. VINTER could also be extended to test traditional block-storage-based file systems without NVM support, bringing features such as automatic atomicity testing over previous work [32]. A generally applicable approach to achieve this would be virtualizing a block storage device and recording a trace of writes as well as flush primitives.

A Artifact Appendix

Abstract

We provide an artifact containing our prototype implementation of VINTER as described in Section 4. The artifact further includes the test cases, configurations, and scripts for reproducing the major parts of our file system analysis (§5) as well as our broader evaluation (§6).

Scope

We aim to achieve two main goals with the artifact. First, it allows reproducing the results of this paper. In particular:

- VINTER can find new bugs in file systems and can help developers with finding the root cause. We provide instructions for reproducing Figures 4 to 6 as well as Section 5.3.
- VINTER can reproduce previously fixed bugs in NOVA. We provide instructions for reproducing Section 6.1.
- VINTER's heuristic is effective at reducing the number of generated crash images without missing semantic states. We provide instructions for reproducing Section 6.2.
- VINTER is sufficiently fast for analyzing file systems. We provide instructions for reproducing Figures 8 and 9.

Second, we provide VINTER for the purpose of analyzing other file systems, in the hope that it will prove useful in developing new NVM file systems.

Contents

Our source code repository (located at `/home/vinter/vinter` in the virtual machine image) contains the following components:

- `README.md` contains general setup information, and `artifact-evaluation/README.md` contains instructions for reproducing the experiments and launching a virtual machine where VINTER is preinstalled.
- `vinter_python/`: The original implementation of VINTER that is used for the analysis in this paper.
  - `pmemtrace.py`: The Tracer component.
  - `trace2img.py`: The Crash Image Generator and Tester components.
  - `trace-and-analyze.sh`: Main script for running the full testing pipeline.
  - `report-results.py`: Script for analyzing output from the testing pipeline.
- `vinter_rust/`: A reimplementation of VINTER in Rust, with the intention of improved performance and to provide a clean base for future extensions.
  - `vinter_trace/`: The Tracer component.
  - `vinter_trace2img/`: The Crash Image Generator and Tester components. Main entry point for running the full testing pipeline.
- `fs-testing/`: Everything related to the analysis of file systems.
  - `scripts/`: Helper scripts, virtual machine (VM) definitions, and test case definitions.
  - `initramfs/`: BusyBox-based user space of the test VMs.
  - `fs-dump/`: File system state extraction program.
  - `linux/`: Configurations of the Linux kernels we test.
- `panda/`: The underlying full system emulator based on upstream PANDA [9] with patches applied.

Hosting

VINTER's source code is available on GitHub at
`https://github.com/KIT-OSGroup/vinter`
on the branch `atc22-artifact`,
commit `4b7e5651e820ec9ebbe2a7321e28b2748103ab74`.

Additionally, we provide an archive containing a virtual machine image that comes installed with VINTER and all its dependencies at doi:10.5281/zenodo.6626098. The archive contains instructions for using the virtual machine image.

References

[1] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

[2] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. "Twizzler: A Data-Centric OS for Non-Volatile Memory." In: *ACM Trans. Storage* 17.2 (June 2021). ISSN: 1553-3077. DOI: 10.1145/3454129.

[3] 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 '16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 83–98. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872406.

[4] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. "FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1077–1091. ISBN: 9781450371025. DOI: 10.1145/3373376.3378515.

[5] 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. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 133–146. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629589.

[6] Luke Craig, Andrew Fasano, Tiemoko Ballo, Tim Leek, Brendan Dolan-Gavitt, and William Robertson. "PyPANDA: Taming the PANDAmonium of Whole System Dynamic Analysis." In: *Workshop on Binary Analysis Research (BAR)*. Vol. 2021. 2021, p. 21.

[7] John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. "Verifying Correctness of Persistent Concurrent Data Structures." In: *Formal Methods – The Next 30 Years*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Cham: Springer International Publishing, 2019, pp. 179–195. ISBN: 978-3-030-30942-8.

[8] 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. Virtual, USA: Association for Computing Machinery, 2021, pp. 503–516. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446744.

[9] 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. Los Angeles, CA, USA: Association for Computing Machinery, 2015. ISBN: 978-1-4503-3642-0. DOI: 10.1145/2843859.2843867.

[10] 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. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592814.

[11] 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. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 100–115. ISBN: 9781450387095. DOI: 10.1145/3477132.3483556.

[12] 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)*. 2020, pp. 652–665. DOI: 10.1109/ISCA45697.2020.00060.

[13] Morteza Hoseinzadeh and Steven Swanson. "Corundum: Statically-Enforced Persistent Memory Safety." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 429–442. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446710.

[14] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. "Log-Structured Non-Volatile Main Memory." In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 703–717.

ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hu.

[15] *IEEE Std 1003.1-2017 (revision of IEEE Std 1003.1-2008) – IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(®)) Base Specifications, Issue 7*. IEEE Computer Society and The Open Group.

[16] Intel. *eADR: New Opportunities for Persistent Memory Applications*. 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[17] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Apr. 2021.

[18] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model." In: *Distributed Computing*. Ed. by Cyril Gavoille and David Ilcinkas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 313–327. ISBN: 978-3-662-53426-7.

[19] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. "Evaluating File System Reliability on Solid State Drives." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 783–798. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/jaffer.

[20] Jungi Jeong and Changhee Jung. "PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency)." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 517–529. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446698.

[21] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. "Crash Consistency Validation Made Easy." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 133–143. ISBN: 9781450342186. DOI: 10.1145/2950290.2950327.

[22] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. "Efficient persist barriers for multicores." In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 660–671. DOI: 10.1145/2830772.2830805.

[23] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. "SplitFS: Reducing Software Overhead in File Systems for Persistent Memory." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 494–508. ISBN: 9781450368735. DOI: 10.1145/3341301.3359631.

[24] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. "SLM-DB: Single-Level Key-Value Store with Persistent Memory." In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 191–205. ISBN: 978-1-939133-09-0. URL: https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet.

[25] 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. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 147–161. ISBN: 9781450368735. DOI: 10.1145/3341301.3359662.

[26] 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. Shanghai, China: Association for Computing Machinery, 2017, pp. 460–477. ISBN: 9781450350853. DOI: 10.1145/3132747.3132770.

[27] 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)*. Philadelphia, PA: USENIX Association, June 2014, pp. 433–438. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz.

[28] 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. Virtual, USA: Association for Computing Machinery, 2021, pp. 487–502. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446691.

[29] 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. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1187–1202. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378452.

[30] 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. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 411–425. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304015.

[31] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging." In: *ACM Trans. Database Syst.* 17.1 (Mar. 1992), pp. 94–162. ISSN: 0362-5915. DOI: 10.1145/128765.128770.

[32] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. "CrashMonkey and ACE: Systematically Testing File-System Crash Consistency." In: *ACM Trans. Storage* 15.2 (Apr. 2019). ISSN: 1553-3077. DOI: 10.1145/3320275.

[33] 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)*. USENIX Association, Nov. 2020, pp. 1047–1064. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/neal.

[34] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. "Reducing NVM Writes with Optimized Shadow Paging." In: *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. Boston, MA: USENIX Association, July 2018. URL: https://www.usenix.org/conference/hotstorage18/presentation/ni.

[35] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. "Memory persistency." In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 265–276. DOI: 10.1109/ISCA.2014.6853222.

[36] 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 USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 433–448. ISBN: 9781931971164.

[37] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. "Persistency Semantics of the Intel-X86 Architecture." In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371079.

[38] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Can Applications Recover from fsync Failures?" In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 753–767. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/rebello.

[39] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. "Programming for Non-Volatile Main Memory Is Hard." In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India: Association for Computing Machinery, 2017. ISBN: 978-1-4503-5197-3. DOI: 10.1145/3124680.3124729.

[40] Steve Scargall. "Introduction to Persistent Memory Programming." In: *Programming Persistent Memory: A Comprehensive Guide for Developers*. Berkeley, CA: Apress, 2020. ISBN: 978-1-4842-4932-1. DOI: 10.1007/978-1-4842-4932-1.

[41] Margo Seltzer, Virendra Marathe, and Steve Byan. "An NVM Carol: Visions of NVM Past, Present, and Future." In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 15–23. DOI: 10.1109/ICDE.2018.00011.

[42] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. "X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors." In: *Commun. ACM* 53.7 (July 2010), pp. 89–97. ISSN: 0001-0782. DOI: 10.1145/1785414.1785443.

[43] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[44] Denys Vlasenko. *BusyBox*. URL: https://busybox.net (visited on 2022-01-10).

[45] 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. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 9781450327046. DOI: 10.1145/2592798.2592810.

[46] 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. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 91–104. ISBN: 9781450302661. DOI: 10.1145/1950365.1950379.

[47] Hu Wan, Youyou Lu, Yuanchao Xu, and Jiwu Shu. "Empirical study of redo and undo logging in persistent memory." In: *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 2016, pp. 1–6. DOI: 10.1109/NVMSA.2016.7547178.

[48] Xiaojian Wu and A. L. Narasimha Reddy. "SCMFS: A file system for Storage Class Memory." In: *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–11.

[49] 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)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 323–338. ISBN: 978-1-931971-28-7. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu.

[50] 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. Shanghai, China: Association for Computing Machinery, 2017, pp. 478–496. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132761.

[51] Jian Yang, Joseph Izraelevitz, and Steven Swanson. "Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks." In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 221–234. ISBN: 978-1-939133-09-0. URL: https://www.usenix.org/conference/fast19/presentation/yang.

[52] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory." In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/yang.

[53] Junfeng Yang, Can Sar, and Dawson Engler. "EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors." In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. Seattle, WA: USENIX Association, Nov. 2006. URL: https://www.usenix.org/conference/osdi-06/explode-lightweight-general-system-finding-serious-storage-system-errors.

[54] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. "EvFS: User-level, Event-Driven File System for Non-Volatile Memory." In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotstorage19/presentation/yoshimura.

[55] Baoquan Zhang and David H. C. Du. "NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Tree with Accumulative Compaction." In: *ACM Trans. Storage* 17.3 (Aug. 2021). ISSN: 1553-3077. DOI: 10.1145/3453300.

[56] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks." In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 207–219. ISBN: 978-1-939133-09-0. URL: https://www.usenix.org/conference/fast19/presentation/zheng.