Silhouette: Leveraging Consistency Mechanisms to Detect Bugs in Persistent Memory-Based File Systems

Bing Jiao
Florida State University

Ashvin Goel
University of Toronto

An-I Andy Wang
Florida State University

Abstract

The emergence of persistent memory (PM), with its non-volatile and byte-addressable characteristics, has led to a novel storage programming paradigm. However, PM programs need to flush stores from CPU caches and correctly order them to avoid inconsistencies after a crash. As a result, many bug-detection tools have been developed for checking crash-consistency bugs in PM software. These bug detectors focus on reordering in-flight stores, crashing the system, and then checking for crash consistency during recovery. However, large-scale systems such as file systems have many in-flight stores, resulting in a large exploration space that makes exhaustive testing prohibitive.

This paper presents Silhouette, a bug-detection framework that targets PM-based file systems. These file systems use standard crash-consistency mechanisms such as journaling and replication. Silhouette uses a novel combination of static instrumentation and data-type-based dynamic analysis to check whether these file systems implement their consistency mechanisms correctly. If these checks pass, then all stores associated with the consistency mechanism (e.g., logging and checkpointing stores for journaling) are considered protected and only the unprotected stores are reordered during exploration. Our evaluation shows that Silhouette dramatically reduces the exploration space, finds all bugs found by existing tools 10x faster, and finds several new bugs in various PM file systems.

1 Introduction

Non-volatile, byte-addressable persistent memory (PM) has garnered attention in recent years due to its low latency and high performance compared to traditional storage media, and high storage density compared to DRAM. As a result, various efforts have focused on developing PM software, including PM indices [5–7, 17, 24, 25, 30, 33, 36, 38, 42–44, 48] and PM file systems [3, 21, 22, 29, 45–47].

However, PM programming is challenging due to its non-volatile and byte-addressable nature, making it prone to bugs, especially in the event of crashes. Stores are not immediately flushed from CPU caches to PM, so a crash may lead to data loss and inconsistency. Furthermore, compilers and processors may reorder instructions for performance, which can lead to crash consistency bugs, e.g., when stores execute out-of-order and a crash occurs in between. Thus PM programs require explicit CPU cache flush and memory fence instructions to persist stores to PM, which complicates these programs and leads to subtle bugs.

As a result, there is a rich body of work and tools for finding bugs in PM software. Yat [27], a seminal exhaustive testing tool, injects failures before fence operations and persists all subsets of in-flight (unpersisted) stores to detect potential bugs. However, this exploration space of all possible post-failure PM states (i.e., crash images) is immense and so exhaustive testing is not feasible. Thus much work has focused on pruning the exploration space to detect bugs in PM libraries and indices [14–16] and PM file systems [23, 28]. Vinter [23] focuses on stores whose data is accessed during recovery, and Chipmunk [28] instruments at cacheline granularity to reduce the search space. However, the exhaustive exploration space remains a challenge.

We propose Silhouette, a framework for detecting crash consistency bugs in PM file systems. Our key observation is that all the PM file systems that we have examined use a set of well-known crash consistency mechanisms, such as journaling and replication, to provide atomicity and durability guarantees. Based on our understanding of these crash consistency mechanisms, Silhouette incorporates a set of invariant checks for these mechanisms. For example, with journaling, an in-place store can only be performed after the store is logged. These invariants help reduce the search space significantly.

Silhouette operates on a file-system execution trace and detects consistency bugs in two steps. First, it applies the invariant checks to determine whether the file system implements its consistency mechanism correctly. To do so, Silhouette uses the store, flush and fence instructions in the execution trace to determine the update time (when the store is issued) and the persist time (when the store is persisted) for each store. The invariant checks use the update and persist time of stores. For example, with journaling, the persist time of the logged store must be earlier than the update time of the in-place store. If any of these checks fail, the file system has a consistency bug. Otherwise, Silhouette considers all the stores associated with the consistency mechanism (e.g., logging and checkpointing stores for journaling) as *protected*. Second, it explores the reordering of stores, similar to previous systems, but only for the unprotected stores, which reduces the search space.

The key challenge in Silhouette lies in implementing invariant checks on an execution trace. While the execution trace contains instructions and memory addresses, the invariant checks are logical, operating on writes to specific data structures (e.g., the head and tail pointers of a log, log entries). Silhouette uses a novel combination of static instrumentation and data-type-based dynamic analysis to implement the invariant checks. We use LLVM instrumentation on file system code to map address ranges to data types in the execution trace. During dynamic analysis, this data type information in the trace enables determining the type (e.g., data structure field type) for PM loads and stores. For example, a store could be mapped to `nova_inode.log_tail = new_tail`, where `nova_inode` is an `inode` type and `log_tail` is the updated field. Silhouette uses lightweight annotations to determine the data structures associated with the invariant checks. For example, the developer specifies that `nova_inode.log_tail` is the NOVA inode's log tail. Silhouette uses these annotations to implement the invariant checks on the execution trace.

We also use a heuristic to reduce the combinatorial testing for unprotected stores. While previous approaches test all combinations of in-flight stores, Silhouette generates only two crash images for each unprotected store. For each unprotected store, we either (1) persist the unprotected PM store but none of the other in-flight stores, or (2) persist all in-flight stores but not the unprotected PM store. These two cases are sufficient for finding all bugs found by existing tools.

We evaluate Silhouette using state-of-the-art PM file systems, PMFS [12], NOVA-fortis (denoted as NOVA in the rest of the paper) [46], and WineFS [21]. Our results show that Silhouette can detect all crash consistency bugs found by two state-of-the-art PM file-system bug-detection tools, Vinter [23] and Chipmunk [28]. In addition, Silhouette finds several new bugs.

The paper makes the following contributions:

- We design, implement, and evaluate Silhouette, a novel framework that leverages knowledge about crash consistency mechanisms to efficiently detect bugs in PM-based file systems.
- We describe the persistence invariants for commonly used consistency mechanisms.
- We propose lightweight annotations for describing the metadata associated with crash consistency mechanisms.
- We present a heuristic for reordering stores that significantly reduces the bug finding exploration space.
- We show that Silhouette significantly reduces the exploration space, finds all existing consistency bugs reported by recent work 10x faster, and finds new bugs.

2 Background and Related Work

In this section, we provide background on persistent memory and then discuss related work on PM bug detection methods to motivate our approach.

2.1 Persistent Memory

PM is a non-volatile storage media technology connected to the memory bus that provides byte-addressable access and 8-byte atomic access, similar to DRAM. Stores are persisted to PM based on the memory persistency model of an architecture [8, 39]. The x86 [20] architecture, for which we implement Silhouette, buffers stores in volatile caches and requires flush instructions (e.g., `flush`, `flushopt`, `clwb`) to persist stores [40], while non-temporal stores bypass caches. The execution of `flush` is ordered with respect to all (both earlier and later) stores regardless of the cache line, while `flushopt` and `clwb` are more efficient and ordered only with respect to earlier stores on the same cache line. Also, stores to the same cache line are persisted in order [27].

To provide control over the order in which stores to different locations are persisted, x86 provides memory barriers such as fence instructions and ensures that the flush instructions cannot be reordered with respect to these fences. Thus fences form *ordering points* at which a store, through a flush and fence, is *persisted*. We refer to a store that has been explicitly flushed and subsequently fenced as a *persisted* store. Otherwise, the store is an *in-flight* store since it may still be in the volatile cache.

2.2 PM Bug Detection

The persistency model described above complicates PM programs, which need to ensure that stores are correctly flushed and ordered or else crashes can lead to incorrect program behavior, including data corruption. Thus many tools have been developed for finding bugs in PM software. We discuss such tools for PM file systems and then for other PM applications.

Yat [27] is an early tool that injects failures before ordering points (fence operations) and persists all combinations of in-flight stores to detect potential bugs. However, this exploration space is immense and so exhaustive testing is

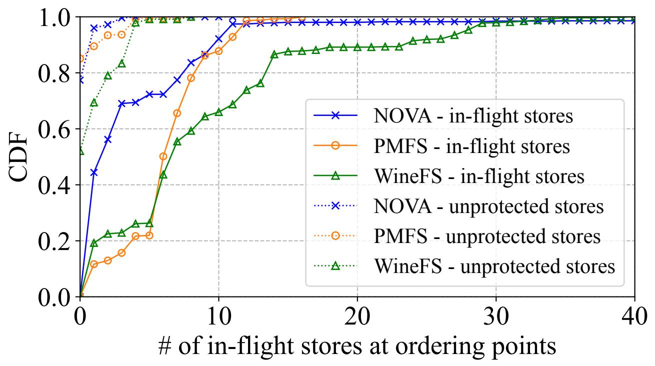

Figure 1: CDF of in-flight and unprotected stores in PMFS, NOVA, and WineFS under the ACE workloads.

not feasible. For example, n in-flight stores (in different cache lines) at an ordering point will generate 2^n crash images. The authors use Yat to find bugs in PMFS [12] and find that some tests would take several years to complete.

Exhaustive exploration is challenging in file systems because of the large number of in-flight stores. Figure 1 shows the cumulative distribution function (CDF) of in-flight stores at ordering points when executing traces of VFS operations from the ACE [35] Seq-3 workload on NOVA [46], PMFS [12] and WineFS [21] (details in Section 5.3).

In this workload, the total number of ordering points for NOVA, PMFS, and WineFS are 13,743, 2,702, and 582 respectively. The large exploration space in PM file systems results from the numerous ordering points, many of which have a large number of in-flight stores. WineFS and PMFS exhibit similar distributions for 50% of the ordering points due to their similar implementations. However, WineFS uses copy-on-write, which requires more transactions for logging block allocation and the extent tree, leading to a longer tail of in-flight stores. NOVA has fewer than 8 in-flight stores for 80% of the ordering points but then has a long tail because it buffers many metadata updates.

Vinter [23] and Chipmunk [28] are state-of-the-art systems for detecting bugs in PM file systems. Vinter uses binary translation to trace PM-related instructions in a virtual machine running unmodified code and reduces the search space by exploring only stores that are accessed during recovery. However, this approach is time consuming because it requires performing recovery to determine the addresses that are read during recovery. Furthermore, its search space is still large and so it was evaluated using only 16 system call sequences. Additionally, this approach may miss bugs, e.g., an ordering point after a missing fence leads to a crash consistent state that requires no recovery. Chipmunk [28] uses Linux Kprobes [26] to instrument flush, fence, and non-temporal store functions. This approach is more efficient than Vinter because it avoids instrumenting at the instruction level but it can miss temporal stores that are not flushed.

The key novelty in Silhouette is its use of data-structure-based analysis to check that the file system implements its crash-consistency mechanism correctly. If so, the stores associated with the consistency mechanism are marked protected. Figure 1 also shows the CDF of (in-flight) unprotected stores. Silhouette explores only these stores, providing better scalability than Vinter and Chipmunk. Furthermore, Silhouette uses the 2CP strategy for exploring unprotected stores, which reduces the search space further.

Witcher [14] uses static and dynamic analysis to analyze program data and control dependencies to infer likely invariants regarding the persistency of program data. Then it generates test cases that violate these invariants and checks whether they cause crash inconsistency in PM indices and key-value stores. Witcher's invariants are based on pairs of memory locations and do not take data structures into account. We ran Witcher on a simplified file system (undo) journal and its recovery code. Witcher did not generate invariants for in-place writes and thus could not check for their correctness.

The large exploration space of in-flight stores has led to an alternative testing approach that avoids exploration and uses manual code annotations to detect typical bugs during program execution [11, 31, 32]. PMTest [32] requires developers to annotate their source code with checking rules to ensure that the code establishes the correct persistency and ordering properties and then evaluates these rules at runtime. XFDetector [31] looks for stores that have not been persisted but are read during post-failure recovery, but it requires various annotations, including to avoid false positives. PMDebugger [11] requires developers to specify durability regions and ensures that stores issued within a region are persisted together. None of these systems have been applied to file system code.

Silhouette's annotations are higher level because they are designed to check the implementation of a crash-consistency mechanism. Unlike code-level annotations, Silhouette's annotations require only specifying the structure names and fields associated with a crash-consistency mechanism.

3 Crash-Consistency Invariants

PM file systems use standard crash consistency mechanisms, such as journaling, to provide crash-consistency guarantees. Using well-known mechanisms simplifies the file system implementation and reduces the potential for bugs.

Based on our understanding of common crash consistency mechanisms, Silhouette incorporates a set of invariants that enable checking whether these mechanisms are implemented correctly [13]. The invariants check that data is persisted in the correct order (*ordering invariants*), in correct locations (*location invariants*) and the correct data is persisted (*data invariants*).

Silhouette operates on a file-system execution trace. The ordering invariants use the store, flush and fence instructions in the execution trace. For each store, we define its *update*

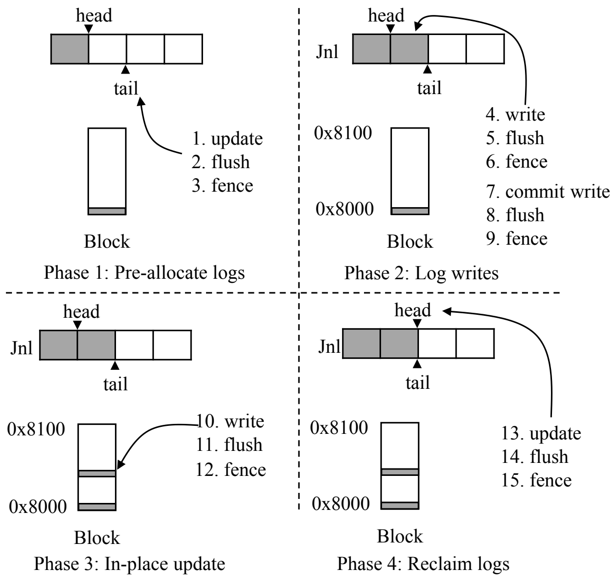

Figure 2: Journaling in PM file systems.

time as the time when the store is issued, its *flush time* as the time when the store is subsequently explicitly flushed, and its *persist time* as the time when a subsequent fence is issued. Two stores are persisted in order when the persist time of the first store *precedes* the update time of the second store.

This section describes three consistency mechanisms, their PM implementations, and their invariants. Section 4 describes how we check these invariants for different file systems using lightweight annotations.

3.1 Journaling

Journaling uses write-ahead logging [34] to record file system updates to a separate journal before persisting the updates in-place to the file system. A typical PM journal implementation uses a head and a tail pointer. The tail points to the first available space at which the next record is written. The head points to the first record that is processed during recovery. When the system is consistent, the two pointers point to the same address. The recovery process scans records from the head to the tail and performs undo or redo processing.

Figure 2 shows that journaling has four phases in PM file systems. Each phase ends with a fence instruction. In this implementation, the first phase pre-allocates the log space by updating and persisting the tail pointer (Steps 1-3). The second phase logs the file system writes starting from the head pointer (Steps 4-6). In addition, some implementations, such as PMFS and WineFS, require committing the log writes (Steps 7-9). The next phase persists the file system writes in place (Steps 10-12). The final phase reclaims the log by persisting the updated head pointer (Steps 13-15).

The flush and fence instructions are critical for ensuring that the stores in each phase are persisted in order. For

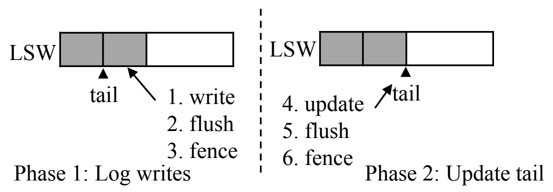

Figure 3: Log-structured writes in PM file systems.

example, without the flush in Step 2 in Phase 1, the in-place write in Step 10 in Phase 3 may be persisted before the log space allocation is persisted, but the system would appear consistent during recovery. The journaling invariants are:

1. Ordering invariants: the persist time of writes in a phase must precede the update time of writes in the next phase.

2. Location invariants: log space is allocated within the valid journal area (typically a circular buffer), and log writes are performed between the head and tail pointers.

3. Data invariants: logged writes have corresponding in-place writes.

3.2 Log-structured Writes

Log-structured writes append file system updates to a log space, a technique used in log-structured file systems [41]. This approach helps avoid random writes and reduces media wear. Generally, log-structured writes use a tail pointer to indicate the first available place to append new data. After a crash, no recovery is needed, since the uncommitted logged data is not visible unless the tail pointer is updated.

Figure 3 shows the phases associated with log-structured writes in PM file systems. The first phase persists the file system writes to the log space (Steps 1-3). Then the second phase updates and persists the tail to commit the updates (Step 3-6). The log-structured write invariants are:

1. Ordering invariants: the persist time of writes in a phase must precede the update time of writes in the next phase.

2. Location invariants: log space is allocated within the valid log area, log writes are performed between the current and updated tail pointers, and log writes do not overlap.

3.3 Replication

Data replication enables recovery from data corruption. For example, if a crash occurs during an update, the recovery process can restore data from an uncorrupted replica (whose content matches its checksum) or when more than a half of replicas have the same content.

Figure 4 shows the phases associated with checksum-based replication in PM file systems. The first step updates and persists the primary replica (Steps 1-4). The data and checksum updates can happen in any order. The second step copies the updates and checksum to the secondary replica (Steps 5-7). The replication invariants are:

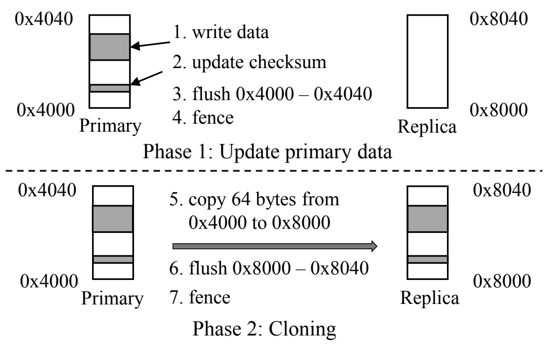

Figure 4: Replication in PM file systems.

1. Ordering invariants: the persist time of writes in a phase must precede the update time of writes in the next phase.

2. Data invariants: replicas should have the same content after they are updated.

4 Silhouette

We propose Silhouette, a scalable testing framework that efficiently detects bugs in PM-based file systems by leveraging the crash-consistency invariants described in Section 3. Figure 5 shows the Silhouette architecture, consisting of three phases, instrumentation, invariant checking and validation. The following sections describe these phases.

4.1 Instrumentation and Tracing

Silhouette uses LLVM to instrument file system code. The instrumented code is linked as a Linux module. When a test case in a workload executes file system code, the instrumentation generates an execution trace as shown in Figure 6. The trace contains instructions executed when top-level file-system-specific VFS operations, such as `pmfs_mount` and `pmfs_create`, are invoked. Silhouette performs invariant checks and validation for each VFS operation invocation *separately*.

The instrumentation uses two passes. The first pass determines the top-level VFS operations by searching the initializer for the Linux `inode` and `file` operations structures. This pass also assigns a unique ID to each LLVM bytecode instruction (not shown in Figure 6) based on the address of the emitted instruction, which helps detect duplicate instruction sequences. Consider the sequence of instructions in a VFS operation. We consider two instruction sequences as duplicates when the PM-related instructions (`store`, `nt_store`, `cas`, `xchg`, `memset`, `memcpy`, `flush` and `fence`) in these sequences have the same IDs. We avoid exploring duplicate instruction sequences in test cases, as explained further in Section 5.3.

The second pass instruments the PM-related instructions. This instrumentation associates a unique timestamp with each instruction in execution order, as shown in the first column of the trace. For all `store`-related instructions, the instrumentation logs the old (before the store) and new (after the store) data, which allows comparing post-recovery file system state with the file system state before and after a VFS operation. We also instrument the *getelementptr* (GEP) instruction, whose operands include the address and type of a data structure, as shown in Figure 6. This instruction computes the address of an element of the data structure and helps identify the data types of stores in the invariant checking phase.

We also parse all the file system source code to determine all data structures, such as the partial `pmfs_journal` and `pmfs_logentry_t` structures shown at the top of Figure 6. Note that the data type column refers to these structures.

4.2 Invariant Checking

The invariant checking phase operates in three steps, as shown in Figure 5. First, it uses the execution trace to determine the data types of all stores. Then, it uses a set of lightweight annotations to identify the stores associated with the crash-consistency mechanisms and tags them with a phase number. For example, journaling has four phases, as described in Section 3. Finally, it checks the consistency invariants to determine whether the crash-consistency mechanism is implemented correctly.

4.2.1 Identifying Data Structure Types

This step takes a single pass over the execution trace to identify the data types associated with writes (i.e., stores, memcpy, etc.), as shown in the data type column in Figure 6. We use the GEP records in the trace to determine these data types. To do so, we maintain an interval tree whose key is an address range and value consists of one or more tuples containing a structure type, field and a timestamp. At each GEP record, we use the GEP data type to update the interval tree for each field of the data type. For example, the GEP record at timestamp 13 in Figure 6 creates entries for each field of the `logentry_t` type, such as `[2000, 2008]: [logentry_t, addr, 13]` and `[2008, 2010]: [logentry_t, size, 13]`. If the data type of an address range changes as a result of a subsequent GEP record, we append a tuple with the new data type and timestamp for that range. We observed two cases where the GEP data type changes in the tested file systems. One case involved a union type, while the other involved variable-length data structures. Then, for writes, we query the interval tree using the address range of the write to determine the data type of the write. If an address range has multiple tuples, we choose the tuple with the largest timestamp that is less than the write's timestamp.

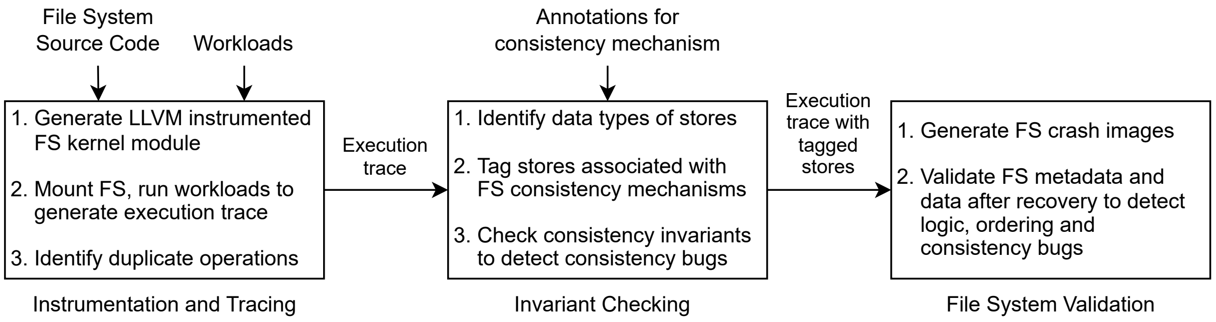
File System Source Code
Workloads
1. Generate LLVM instrumented FS kernel module
2. Mount FS, run workloads to generate execution trace
3. Identify duplicate operations
Instrumentation and Tracing
Execution trace
Annotations for consistency mechanism
1. Identify data types of stores
2. Tag stores associated with FS consistency mechanisms
3. Check consistency invariants to detect consistency bugs
Invariant Checking
Execution trace with tagged stores
1. Generate FS crash images
2. Validate FS metadata and data after recovery to detect logic, ordering and consistency bugs
File System Validation

Figure 5: The Silhouette architecture.

Metadata	Annotation
Journal head	pmfs_journal.base + pmfs_journal.head
Journal tail	pmfs_journal.base + pmfs_journal.tail
Buffer addr	pmfs_journal.base
Buffer size	pmfs_journal.size
Dest addr	pmfs_logentry_t.addr_offset
Dest size	pmfs_logentry_t.size
Pre-allocate	true
LSW tail	nova_inode.log_tail
LSW log size	4096
LSW log links	nova_inode_page_tail.next_page
Pre-allocate	false
Replication structures	nova_inode, nova_file_write_entry, nova_dentry, nova_setattr_logentry, nova_link_change_entry
# of replicas	2

Table 1: Annotation for PMFS's journal (WineFS uses the same annotations), NOVA's log-structured write (LSW) and replication. Annotations for NOVA's journal are not shown.

4.2.2 Lightweight Annotations

While the previous step identifies the data structure and fields associated with writes, we need to know the file-system structures that are associated with the crash consistency mechanisms. We use a small set of programmer-specified annotations to determine these data structures. The annotations are specified in a separate configuration file and require no modifications to the file system code. Table 1 shows the annotations for PMFS's journal, NOVA's log-structured writes and NOVA's replication. These annotations and NOVA's journal annotations are sufficient for checking all the invariants implemented in Silhouette.

The PMFS journal data structure `pmfs_journal` has the `base`, `head`, and `tail` structure fields, as shown in Figure 6. In PMFS, the journal is a circular buffer but the head and tail are offset addresses. The journal head and the tail pointers are computed by adding the base address to the head and tail offsets, as shown in the Journal head and tail annotations (first and second rows). [1] The Buffer addr and size (third and fourth rows) specify the journal area. They help with detecting wrap around in the circular buffer. The PMFS journal entry data structure `pmfs_logentry_t` is shown in Figure 6. The Dest addr and size annotations (fifth and sixth rows) specify the address of the in-place write and the size of the logged data, which help determine the location of the in-place writes. The allocation annotation (the seventh row) indicates that PMFS pre-allocates log space, as shown in the first phase of Figure 2.

[1]The annotation language supports simple arithmetic operations.

For NOVA's log-structured writes, the LSW tail annotation (first row) specifies the tail pointer. NOVA uses a linked list for its log-structured writes. The log size and links annotations (second and third rows) enable traversing the log. NOVA's log-structured writes are per-file and thread-safe, so it does not pre-allocate log space (fourth row). NOVA also uses replication to protect the per-file logs.

For replication, we only need to know the data structures that have replicas, since any writes to the data structure must have corresponding writes to its replicas. We also require the number of replicas to determine when all the replicas are synchronized.

4.2.3 Identifying Phases of Consistency Mechanisms

This step uses the annotations to identify the writes associated with crash-consistency mechanisms and tags them with a phase number, as shown in the tag column in Figure 6. Next, we describe this process for each crash-consistency mechanism.

Journaling: We use two passes over the execution trace to tag stores with the four phases associated with journaling. The first pass searches for stores with a data type that matches the journal structure (e.g., `journal`). In Figure 6, these are stores at Records 3-5, 10, and 23. Stores 3-5 are issued by the mount function and so they are tagged as initialization stores. Stores 10 and 23 are tagged Phase 1 and Phase 4 because they match the head and tail annotations.

This pass also searches for stores with a data type that matches the journal entry type (e.g., `logentry_t`). In this case, they correspond to Records 14-16 and they are tagged Phase 2. We use the Dest addr and size annotations to determine the range of in-place updates from log entries. For instance, Record 14 has an address value `0x8000` and Record

```
struct pmfs_journal {
    u64 base;
    u32 head;
    u32 tail;
};
```

```
struct pmfs_logentry_t {
    u64 addr_offset;
    u64 size;
    char[48] data;
};
```

time	inst	addr (hex)	size (dec)	old value (hex)	new value (hex)	data type	tag
1	startCall	pmfs_mount					
2	gep	1000	N/A			journal	
3	store	1000	8	0	2000	journal.base	journal.init
4	store	1008	4	0	0	journal.head	journal.init
5	store	100C	4	0	0	journal.tail	journal.init
6	flush	1000	16	N/A			
7	fence	N/A					
8	endCall	pmfs_mount					
9	startCall	pmfs_create					
10	store	100C	4	0	40	journal.tail	journal.p1
11	flush	100C	4	N/A			
12	fence	N/A					
13	gep	2000	N/A			logentry_t	
14	store	2000	8	0	8000	logentry_t.addr	journal.p2
15	store	2008	8	0	30	logentry_t.size	journal.p2
16	memcpy	dst: 2010 src: 8000	48	xx	xx	logentry_t.data	journal.p2
17	flush	2000	64	N/A			
18	fence	N/A					
19	store	8000	48	xx	xx	xx	journal.p3
20	flush	8000	48	N/A			
21	fence	N/A					
22	store	3000	8	xx	xx	xx	unprotected
23	store	1008	4	0	40	journal.head	journal.p4
24	flush	1008	4	N/A			
25	fence	N/A					
26	endCall	pmfs_create					

Figure 6: A simplified PMFS execution trace generated by the instrumentation phase. The gray fields are filled at run time in the invariant checking phase.

15 has a size value `0x30`. We track this in-place update range `[0x8000, 0x8030)`.

The second pass finds stores that lie within any in-place update range and tags them as Phase 3, e.g., Record 19.

Log-structured writes: We use two passes over the execution trace to tag stores with the two phases associated with log-structured writes. The first pass searches for stores with the tail data type (e.g., `nova_inode.log_tail` in Table 1). These stores are tagged Phase 2. Then we extract the old and new values of the tail update, which forms the log space address range. The second pass looks for stores that lie within any log space address range and tags them as Phase 1.

Replication: We use two passes over the execution trace to tag writes with the two phases associated with replication. Currently, we assume that replicas are synchronized using the `memcpy` instruction. The first pass searches for `memcpy`

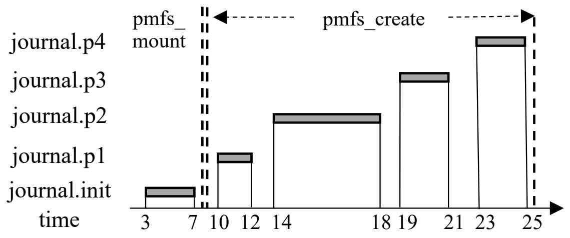

Figure 7: The horizontal bars show the in-flight period of writes in different journaling phases.

with a data type that matches the replication annotation and tags them as Phase 2. It also records the source address range of the instruction. The second pass looks for stores that lie within any source address range and tags them as Phase 1. We also support `memcpy`-based replication that is performed by copying from PM to DRAM and then to PM.

Unprotected writes: All remaining untagged stores are then tagged as unprotected stores.

4.2.4 Checking Invariants

Our invariant checks require the flush and persist time for all stores associated with the consistency mechanisms. We derive these times by modifying Witcher's cache/NVM simulator that simulates the effects of store, flush and fence instructions as per the memory consistency model of the x86-64 architecture [14]. When a store is issued to a cache line, we append it to the cache line with null flush and persist times. When a flush is issued on a cache line, all stores in the cache line that have a null flush time are assigned a flush time. When a fence is issued, stores in all cache lines with a non-null flush time are assigned a persist time, and removed from their cache line, indicating that they are persisted.

We call the time interval of an in-flight store, from update time to persist time, the *in-flight period*. Similarly, the in-flight period of a phase is the earliest update time and latest persist time of the stores in the phase. The invariant checks typically traverse the execution trace, look for stores associated with a consistency mechanism in timestamp order, and check ordering invariants based on these in-flight periods. Next, we describe how we check invariants for each consistency mechanism.

Journaling: Figure 7 shows the in-flight period of the journaling phases for the execution trace shown in Figure 6. The ordering invariants check that the in-flight periods of the different phases occur in order and are not overlapping. For the location invariants, when we find a Phase 1 store (tail update, T10-12), we use the address range from the old value to the new value of the tail to check that the log writes (Phase 2, T14-18) are within this address range. For the data invariants, when we find a Phase 4 store (head update, T23-25), we check whether Phase 2 log writes have matching Phase 3 in-place writes and with the same content.

Log-structured writes: The invariant checks for log-structured writes are similar to the journaling invariants. The ordering invariants simply check that the Phase 1 in-flight period is earlier than the Phase 2 in-flight period. The location invariants check that the log writes do not overlap and are performed in the valid log area.

Replication: The ordering invariants check that the Phase 1 in-flight period is earlier than the Phase 2 in-flight period. The data invariants check that the contents of the Phase 1 writes match the Phase 2 `memcpy` operation. While we check consistency invariants for each VFS operation separately, some VFS operations, e.g., `nova_mkdir`, may replicate multiple objects in overlapping in-flight periods. The replication invariants operate on each object separately.

4.3 File System Validation

After the invariant checking phase, the file system validation phase in Silhouette tests for crash consistency. It aims to validate whether a file system remains consistent, i.e., provides its stated consistency guarantees, in the presence of crashes. While the POSIX standard does not define file-system crash consistency semantics [2], the PM file systems we analyzed provide strong atomicity guarantees for all metadata and file data operations. Thus the validation phase checks whether the file system provides operation atomicity (i.e., all-or-nothing semantics) after a crash.

Our testing methodology is based on constructing and testing file-system crash states using oracles, which avoids false positives, similar to previous approaches [14, 23, 28]. However, our search space is more targeted, and we perform detailed consistency checks on the crash states, both of which help reveal new file system bugs. Next, we describe how we generate and then test file-system crash states.

4.3.1 Generating Crash States

For testing, we reuse the execution trace generated by a test case and simulate a crash before each reordering point within a file system operation in the trace. At each crash, we generate *crash plans*, which consist of all the persisted writes (writes with a persist time that precedes the crash time) and a subset of in-flight writes since they may not have been persisted.

We generate crash plans targeting 1) crash-consistency mechanisms, and 2) unprotected stores. The former detect crash-consistency mechanism bugs, while the latter detect ordering and other logic bugs (bugs that are not fixed by adding flush or fence instructions).

Crash-consistency mechanisms: The consistency invariants we use may not be comprehensive, and the checks could have been implemented incorrectly, both of which would lead to missing bugs. Thus, we generate crash plans to detect potential crash-consistency mechanism related bugs during recovery. For journaling, Silhouette generates a crash plan before Phase 4 (writes in Phases 1, 2, and 3 are persisted) to trigger journal recovery. For replication, Silhouette generates two crash plans: 1) writes to the primary are partially persisted (e.g., Step 1 is persisted but Step 3 is not persisted in Figure 4) to test whether the primary can be recovered from the secondary, and 2) writes to the secondary are partially persisted (e.g., parts of `memcpy`) to check whether the secondary is updated from the primary during recovery. For log-structured writes, Silhouette does not generate crash plans since the tail updates are performed atomically.

Unprotected stores: Generating file-system crash plans based on reordering in-flight stores [23] or cache lines [28] leads to an exponential number of plans (2^n plans for n in-flight stores or cache lines). Figure 1 shows that PM file systems have as many as 40 in-flight stores at reordering points, which exacerbates this problem.

Silhouette generates crash plans based only on the unprotected (in-flight) stores. Figure 1 shows that the maximum number of such stores is fewer than 10, which significantly reduces the search space.

However, instead of exploring all combinations of unprotected stores, we propose a simple heuristic that only generates two crash plans for each unprotected store. Thus, with n unprotected stores, we generate $2n$ crash plans. For each unprotected store, the first crash plan persists only the chosen unprotected store, while none of the other in-flight stores are persisted, and the second crash plan, persists all in-flight stores except the chosen unprotected store.

This 2CP (two crash plans) scheme detects all file-system bugs found by Vinter and Chipmunk for the three file systems we use in our evaluation. The reason is that PM programs often use a single *critical* store for persistence ordering. For example, a guarded read during recovery of the form "`if (flag) return data;`" likely implies that the persist time of `data` is earlier than the update time of the critical `flag` variable [14]. Below, we denote this persists-before relation as `data`→`flag`.

Our 2CP scheme can detect violations of all persists-before relations of the form `(S1, S2, ...)`→`C`, and `C`→`(S1, S2, ...)`, where `C` is the critical store, and `(S1, S2, ...)` are one or more other stores. Assuming that an unprotected store is a critical store, our first crash plan, which only persists the unprotected store, detects violations of the first persists-before relation, and our second crash plan, which persists all the other stores, detects violations of the second persists-before relation. Since we do not know whether an unprotected store is a critical store, we generate two crash plans for each unprotected store (thus treating each unprotected store as a critical store).

The 2CP scheme will not exhaustively test cases such as `(A, B)`→`(C, D)`. We have only observed one real-world case with this constraint. Consider checksum-based replication in which if the primary is corrupted, it can be recovered from the secondary, which has the correct checksum. However, if there is no fence between the updates to the two replicas then the persist-before relationship does not exist. In this case, even though the replicas could be corrupted, (e.g., only `A` and `C` are persisted), 2CP will not construct this crash plan. Fortunately, our replication invariants test this case.

4.3.2 Testing Crash States

For each crash plan, we create a *crash image*, which is a file system image that is constructed by replaying the writes in the crash plan. To test a crash image, we mount the image, which runs file-system recovery code, and check the system log for any errors (e.g., kernel panic, KASAN report), which we call the *syslog test*.

Then we compare the recovered file-system state against the file-system states *before* and *after* the file-system operation (at which the crash occurred in the crash plan) has run. These pre- and post-operation states are obtained via `stat` when running the test cases. A successful match against either of them indicates that the operation executed atomically and durably, otherwise a bug is found. We compare the recovered file system against the pre- and post-operation states by checking the `stat` metadata for all the files and directories in the file system, which we call the *stat test*.

Some PM programs implement lazy recovery mechanisms to improve their performance. For example, NOVA rebuilds a file's extent tree only when it is accessed. To ensure that the lazy recovery works correctly and the recovered file system is usable, we check that all the files and directories are writeable and removable, which we call the *write test*.

We found that the pre/post-operation state may not be reliable due to logic bugs. For instance, the recovered state may match the pre-operation state but internal metadata not exposed by `stat` may be modified, or the file-system state may not reflect the expected state after the file system operation. As a result, we created two additional tests compared to previous work. In the *unprotected store test*, if the recovered file system matches the pre-operation (post-operation) state, then for 2CP, we test that the data at the chosen unprotected store matches the old (new) value of the store. In the *file operation test*, we use operation-specific checks to verify the expected behavior of some file operations. For example, after an `append` operation, the file size should increase by the written size.

5 Evaluation

This section evaluates Silhouette by first presenting the bugs that we found in three PM file systems, PMFS [12], NOVA-fortis [46], and WineFS [21]. Then, we compare the bug found timing of Silhouette with Vinter [23] and Chipmunk [28], two state-of-the-art systems that focus on detecting bugs in PM-based file systems. Last, we evaluate the scalability of Silhouette.

Implementation: Silhouette uses a client-server architecture. Each client is a VM that runs a test case, checks invariants, generates crash plans and performs recovery. To do so, we leverage Witcher's cache simulator framework [14]. The server runs on the host and manages the VMs, including monitoring and restoring their state. As mentioned in Section 4.1, Silhouette's static instrumentation assigns unique IDs to instructions in an instruction trace. The server stores a hash of the IDs of the PM-related instructions (e.g., PM store, flush, fence etc.) in each operation in the execution trace. When two invocations of an operation (in the same or different test cases) have the same hash, we only explore the operation once. Silhouette randomly samples `memcpy` and `memset` and treats the multiple stores issued by them as a single atomic store. Silhouette is open-sourced at `https://github.com/iaoing/Silhouette`.

System setup: All file systems are built as Linux v5.1 kernel modules. We used the file system versions that were used by Chipmunk in their evaluation. We enabled the checksum, parity, and CoW options in NOVA. Also, we enabled the CoW mode in WineFS because it ensures atomicity for data operations. Our evaluation uses QEMU [1] VMs, running on a Dell 7820 host machine. This machine is equipped with an Intel Xeon Silver 4215R CPU, 144 GB DRAM (128 GB simulated by a 128 GB Intel Optane PM 100 Series), Western Digital 2 TB HDD (containing the VM images). Each QEMU VM is configured with 1 core, 8 GB DRAM, and a 128 MB PM device (emulated by DRAM). Both the host and the VMs run Ubuntu 20.04.4 LTS.

Workloads: We used the ACE workload generator [35], which generates test cases using a sequence of VFS operations with the correct prerequisite operations (e.g., `open`) and specified sets of parameters. We tested the following 11 operations: `creat`, `mkdir`, `fallocate`, `write`, `symlink`, `link`, `unlink`, `remove`, `rename`, `truncate`, and `rmdir`. We omitted the operations that are irrelevant for PM (e.g., `fsync`, `flush`). We also explored some custom workloads (e.g., to exercise long file names). Finally, we created NOVA workloads to test NOVA's functionality (e.g., large log-structured writes, and `create_snapshot` and `delete_snapshot` operations).

5.1 Bug Analysis

Silhouette found all bugs reported by Vinter [23] and Chipmunk [28]. Vinter and Chipmunk reported 7 and 20 bugs for the three PM file systems. In addition, Silhouette found 15 previously unreported (new) bugs, as shown in Table 2. We have filed these bugs and currently 3 have been confirmed and fixed.

The *Type* column in Table 2 shows that 1 bug is a PM

Bug	FS	Type	Cause	Effect	Test
1	NOVA	PM	Replica pointer not persisted correctly during inode allocation	Segfault	syslog
2	NOVA	Logic	`i_size` and `i_blocks` fields not set correctly in `symlink`	Inconsistent attributes	stat
3	NOVA	Logic	`i_blocks` field not set correctly in `fallocate`	Inconsistent attributes	unprotected store
4	NOVA	Logic	`truncate` is not atomic	Data leak	unprotected store
5	NOVA	Logic	Different `dentry`s have the same `inode` number	Incorrect `inode number`	file operation
6	NOVA	Logic	Atomicity violation in `Unlink` and `rmdir`	File/Dir is inaccessible	write
7	NOVA	Logic	Snapshot ID set incorrectly during recovery	Cannot create snapshots	write
8	NOVA	Logic	Traversing snapshots fails after snapshot removal	Various errors	syslog
9	NOVA	Logic	Extent tree unreadable due to wrong checksum	Various errors	syslog
10	NOVA	Logic	Unsafe user space read in procfs	Segfault	syslog
11	NOVA	Logic	DRAM inode structure not initialized	Segfault	syslog
12	PMFS	Logic	`O_APPEND` doesn't work correctly	Data loss	file operation
13	PMFS, WineFS	Logic	Reuse `inode` in orphan list	Data loss	stat
14	NOVA	Perf	Garbage collection information not updated atomically	Incorrect GC trigger	unprotected store
15	All	Perf	Redundant memory barriers	Performance degradation	-

Table 2: New bugs found by Silhouette.

bug; 12 are logic bugs; and 2 are performance bugs. PM bugs can be fixed by adding flush or fence instructions while logic bugs are crash-related bugs in program logic that would not be fixed by simply adding flush or fence instructions. Performance bugs may degrade system performance. The *Cause* and *Effect* columns show the root cause and the effects of the bug. The *Test* column shows the Silhouette test (see Section 4.3.2) that detected the bug. Next, we describe a few bugs that were detected using different tests.

Bug 1: Replica pointer not persisted correctly during inode allocation. NOVA uses a per-inode log page (`inode->page`) and a replica of this page (`inode->replica_page`) to log inode updates. The log page also maintains a pointer to the replica page (`page->replica`). This bug occurs when an inode's log page is allocated but the log page's replica pointer is not persisted before the inode (including `inode->page`, `inode->replica_page`, and the inode's checksum) is persisted. After a crash, the recovery code checks the inode's checksum and this check passes. However, during normal operation, when file operations update the inode, NOVA accesses the inode's log page and may access its replica pointer, which is null and causes a crash. This bug can be fixed by adding a fence after the replica pointer is flushed.

Silhouette generates a crash plan that persists the inode but not `inode->page->replica`. It detects this bug because after recovery, we write to the file, which causes a segmentation fault. Vinter fails to detect this bug since a log page's replica pointer is not read during recovery and so Vinter does not create a crash plan to test it. Chipmunk's also does not detect this bug because its consistency check does not access the page's replica pointer.

Bug 2: Inconsistent `inode` attributes between PM and VFS cache. The `symlink` operation in NOVA updates the inode's `i_size` field in PM and the VFS cache inconsistently (there is an off-by-one error). After an inode is evicted from the VFS cache or after a remount, this field is read from PM and updated in the VFS cache, which then has consistent values. We detect this bug because the recovered PM state shows that the `symlink` operation is complete but the file size attribute is different between the recovered state and the post-operation state. Chipmunk did not find this bug because it did not test the `symlink` operation. Vinter tests `symlink` but did not report this bug.

Bug 4: `truncate` is not atomic. When truncating a file to a smaller size, the truncated data should be erased or else it would be accessible if the file is subsequently truncated to a larger size. In the `truncate` operation, NOVA changes the file size and then erases the truncated data but these operations are not performed atomically. The erasure is conducted using a `memset` instruction (unprotected stores). If a crash occurs before the erasure, then after recovery, Silhouette reports that the attributes of the recovered file system match the post-operation state but the unprotected store data matches the pre-operation state. Chipmunk and Vinter did not find this bug because they compare file-system attributes but not the file-system state.

Bug 5: Different `dentry`s have the same inode number. After creating a new file, Silhouette checks whether the inode number of the file is unique. This file operation test revealed that NOVA, in some cases, generates duplicate inode numbers, where two files (with different directory entries) have the same inode number. This bug occurs in the `readdir` operation. Vinter and Chipmunk did not find this bug because they do not perform the unique inode check.

Bug 11: DRAM inode structure not initialized. During mount, NOVA reads a PM file to recreate its inode allocator information in DRAM. To speed up recovery, it caches the PM

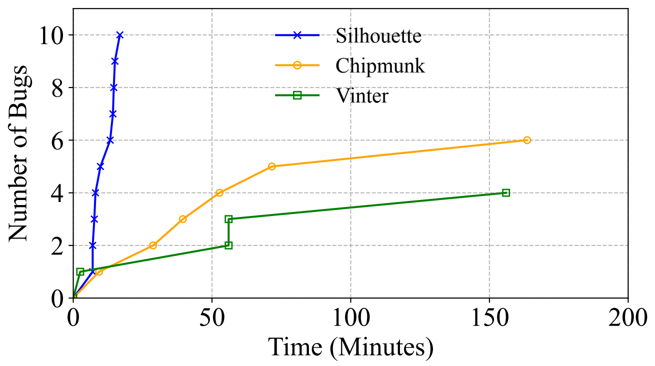

Figure 8: Bugs found in Nova with the ACE Seq3 workload.

file's inode in a temporary DRAM inode structure. However, it does not zero out this inode structure. After the inode allocator information is rebuilt, NOVA accesses the DRAM inode structure to clear the PM file. If the inode structure has garbage data, a segmentation fault occurs when NOVA dereferences a field in the structure. Silhouette detected this bug because it generated a crash plan for the `unmount` operation, which caused a crash during recovery. However, this crash does not occur on a mount after a clean unmount. Vinter and Chipmunk did not find this bug because they did not test the `unmount` operation.

Bug 13: Reuse `inode` in orphan list. When truncating or unlinking an open file (or directory), PMFS and WineFS add an entry consisting of the file's inode number and the new file size (0 for `unlink`) to a persistent orphan list (similar to Ext4). If a crash occurs while the file is open, the recovery process traverses the orphan list to set the size of truncated files based on their recorded size and to reclaim unlinked inodes. During normal operations, when the truncated or unlinked file is finally closed, the VFS code invokes a file system function asynchronously to reclaim the inode (if `nlinks` count is zero) and remove its corresponding entry from the orphan list. However, a newly created file or directory may be assigned the same inode number after the original inode is freed but before its orphan list entry is removed by the asynchronous function. If a crash occurs before the orphan list entry is removed, then the recovery process will find the inode of the newly created file in the orphan list entry. Since the `nlinks` count of this file is greater than zero, the file size is truncated to 0, causing data loss. Silhouette found this concurrency bug because it did not replay the asynchronous PM writes (removal of the orphan list entry) when generating crash plans and detected a directory of size 0 after recovery.

5.2 Bug Finding Time

We compare the time it takes Silhouette, Chipmunk, and Vinter to find bugs in NOVA. For this test, we used the NOVA version evaluated in Chipmunk's study since Chipmunk did not report any new bugs in the latest version of NOVA. We ran the ACE Seq3 workload on one VM and limited the test to 6 hours since no bugs were found beyond this time.

Figure 8 shows the bugs found over time. Vinter only tested 36 test cases in 6 hours and found 4 bugs in 156 minutes. Two bugs caused by atomicity violations in `rename` were previously reported by Vinter. An atomicity bug in `link` and `unlink` and a missing inode checksum update when initializing LSW's logs were previously reported by Chipmunk. Chipmunk tested 2.6K test cases in 6 hours and found 6 bugs in 164 minutes, including the 4 bugs found by Vinter. The two additional bugs are a missing memory fence after updating a metadata checksum, and a failure to atomically update data and its parity during `truncate`, both of which were reported by Chipmunk.

Silhouette tested 5.3K test cases in 6 hours and found 10 bugs in 17 minutes, including the 6 bugs found by Chipmunk and 4 new bugs (Bugs 3, 5, 14, and 15). The rest of the bugs in Table 2 were not found since we only tested operations handled by all three systems. Thus, we did not test symlink (Bug 2), snapshots (Bugs 6, 7, 8, and 10), and mount (Bug 11). Additionally, we disabled Bug 9 since it corrupts the file system, preventing further testing of other operations. Bugs 1 and 4 were only found in later NOVA versions and they are triggered by subsequent bug fixes.

5.3 Silhouette Scalability

In this section, we evaluate the scalability of Silhouette by comparing the number of in-flight stores and crash plans generated by Silhouette, Vinter and Chipmunk. To measure scalability, we used the Seq3 workload for these experiments to evaluate all systems. Similar to Chipmunk, we sampled 50K cases, as described under Workloads in Section 5.

For Silhouette, we ran 10 VMs in parallel and the total run time is roughly 6.8, 3.4, and 5.1 hours for NOVA, PMFS, and WineFS. Due to Chipmunk's lack of support for parallel testing and Vinter's slow running speed, we disabled their consistency checks (e.g., constructing crash images and running recovery) so they can be run in reasonable time.

As mentioned in Section 4.1, Silhouette's static instrumentation assigns unique IDs to instructions, which enables detecting duplicate instruction sequences in an execution trace. We explore duplicate sequences once, whether in the same or different test cases. For example, suppose Test Case 1 has operations `(A1, B1)` and Test Case 2 has operations `(A1, B2)`, where `A` and `B` are two different types of operations, and `A1`, `B1` and `B2` are three unique instruction sequences. For Test Case 1, we explore the instructions in `A1` and `B1`. However, for Test Case 2, we *only* explore the instructions in `B2` after persisting all the writes in `A1`. Although the 50K test cases in the Seq3 workload contain 391,743 operations, Silhouette identified only 356, 285 and 81 unique operations for NOVA, PMFS and WineFS. NOVA uses dedicated log types for different operations (e.g., link change log, data write log), leading to more unique operations. WineFS has fewer unique operations compared to PMFS since its CoW strict mode leads to more code reuse.

Vinter's crash plans do not contain instruction addresses and so it cannot detect duplicate instruction sequences accurately. For example, two similar store sequences for an operation may be emitted from different instructions. Chipmunk only instruments flush and fence functions (but not stores) and so it also cannot perform duplicate detection. To assess the benefits of duplicate detection in these systems, we used Silhouette to also generate test cases that avoid exploring duplicate operations more than once and evaluated Vinter and Chipmunk using them. We refer to these test cases as Vinter/Chipmunk *unique* test cases.

Number of in-flight stores: Silhouette considers unprotected (in-flight) stores, Vinter considers all in-flight stores and Chipmunk considers in-flight flush operations when generating crash plans. Figure 9 shows their CDF.

The Silhouette plots for the unprotected stores are the same as the plots shown in Figure 1. While Vinter only reorders in-flight stores whose data is read during recovery, the number of such in-flight stores is still comparable to the total number of in-flight stores, as shown in Figure 1. Vinter's approach can miss bugs, e.g., if a fence is missing but the next fence leads to a crash-consistent state that requires no recovery. Chipmunk's in-flight flushes are comparable to Silhouette's unprotected stores. Unlike Silhouette, Chipmunk only observes stores that are flushed. Moreover, if two stores write to the same cache line and are then flushed, then Chipmunk cannot explore an execution sequence in which only the first store is persisted.

Number of crash plans: Silhouette generates crash plans for checking the crash-consistency mechanisms and for reordering the unprotected stores based on the `2CP` scheme. Vinter generates crash plans by considering all subsets of in-flight store operations whose data is read during recovery, but it roughly limits the number of crash plans per ordering point to 20. Chipmunk generates crash plans by considering all subsets of in-flight flush (not store) operations but limits the number of crash plans per ordering point to $F*(F-1)/2$, where F is the number of in-flight flush operations. Silhouette does not impose any limit on the number of crash plans generated by the `2CP` scheme.

Table 3 shows the number of crash plans generated by Vinter, Chipmunk, and Silhouette for the ACE Seq3 workload. For Silhouette, we break down the number of crash plans for checking the crash-consistency mechanisms and the unprotected stores. The number of crash plans generated by Chipmunk and Vinter ranges between 1-3 million. In contrast, Silhouette generated 100x - 3000x fewer crash plans because it avoids exploring duplicate operations, performs invariant checking, and uses `2CP` exploration.

Compared to Vinter (unique) and Chipmunk (unique), Silhouette generates 1.9x and 4.2x fewer crash plans on NOVA, 11x and 6.3x fewer crash plans on PMFS, and 7.3x and 2x fewer crash plans on WineFS. Unlike Vinter and Chipmunk, which arbitrarily limit the number of crash plans to prune the exploration space, Silhouette reorders all unprotected stores. Further, Silhouette is much more scalable than the (original) Vinter and Chipmunk and thus we are able to run many more test cases (e.g., more types of system calls, test snapshots, etc.) and we can perform more detailed (unprotected store and file operation) tests on the crash states.

Scheme	NOVA	PMFS	WineFS
Vinter	1,928,524	2,375,295	3,312,029
Chipmunk	3,392,143	1,640,534	995,865
Vinter (unique)	27,931	26,645	7,860
Chipmunk (unique)	61,218	15,386	2,179
Silhouette	6,378+8,038	265+2,162	61+1,018
2CP	115,304	26,472	5,888
Invariants+Comb	6,378+18,088	265+12,956	61+6,842

Table 3: The number of crash plans generated by Vinter, Chipmunk and Silhouette for the ACE seq3 workload.

Next, we evaluate invariant checking and `2CP` reordering individually. First, we disabled invariant checking and applied `2CP` reordering for all stores. The `2CP` row in Table 3 shows that disabling invariant checking generates 5-11x more crash plans than Silhouette. Second, we used invariant checking but used exhaustive exploration, instead of `2CP`, for all unprotected stores. The `Invariants+Comb` row shows that exhaustive exploration generates 2.3-6.7x more crash plans than `2CP` reordering, which is reasonable since the number of unprotected stores at ordering points is relatively small, as shown in Figure 9.

While the `2CP` scheme generates more crash plans than Silhouette, it does not require crash-consistency invariants. We found that `2CP` was able to find all the new bugs that we found in Table 2. Thus, `2CP` can serve as a good starting point for detecting crash consistency bugs. Then, adding the invariant checks further reduces the search space.

5.4 Discussion

Concurrency operations: Similar to prior works, Silhouette does not support concurrent workloads. Although Bug 13 is triggered by a concurrent (asynchronous) operation, Silhouette found it by not persisting the asynchronous PM writes. Also, detecting bugs in concurrent operations requires finer-grained control over concurrent accesses and thread interleaving, which we leave as future work.

False positives: Silhouette can produce false positives because the mismatched data content detected by the *unprotected store* test may not affect file system consistency. For example, the data may be in unallocated or unreferenced blocks. Therefore, manual verification is needed. Fortunately,

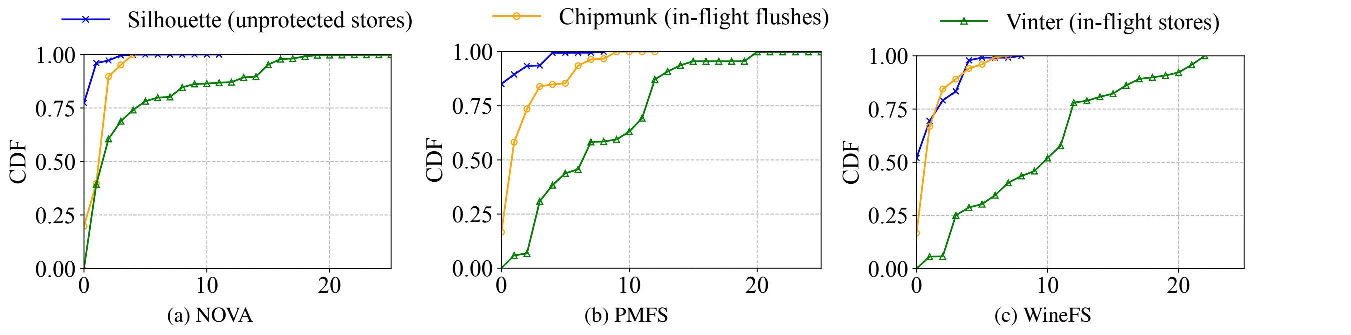

Figure 9: CDF of unprotected stores for Silhouette, in-flight stores for Vinter and in-flight flushes for Chipmunk at ordering points in the Seq3 workload.

not much effort is needed since most writes were tagged with the data type (structure fields). After identifying one false positive, all cases with the same data type can be excluded.

False negatives: Silhouette may miss bugs since the 2CP method tests only one PM critical store at a time and cannot handle the persistence ordering such as `(A, B)`→`(C, D)` that we found for checksum-based replication. Nevertheless, the 2CP method can be extended to test pairs of PM critical stores (e.g., both `A` and `C`).

Annotations: Our annotation-based approach requires some developer effort. However, the annotations can be provided incrementally since they are optimizations that help prune the search space. For example, if only the head and tail pointers are known, Silhouette can still detect Phases 1, 2, and 4 for journaling and apply invariant checks on them. The in-place updates in Phase 3 will not be recognized but these unprotected writes will be checked by our 2CP scheme.

Deploying Silhouette for a new PM file system involves work for the file system developer and possibly for the Silhouette developer. If the new file system uses one of the crash-consistency mechanisms described in Section 4, then the file system developer needs to add annotations that specify the structure fields shown in Table 1. For example, when we moved from NOVA to PMFS, we noticed that PMFS uses preallocation for logging, but other than that, it took a graduate student roughly 30 minutes to annotate PMFS. If a file system uses a new crash-consistency mechanism, then the Silhouette developer will need to define the update phases and invariants of the crash-consistency mechanism. However, in our experience, file systems generally use the well-tested mechanisms that are supported by Silhouette.

Extended ADR: With the introduction of extended asynchronous DRAM refresh (eADR) [18] in Intel® x86 architectures, CPU caches are persisted during crashes, obviating the need for flush instructions. For example, eADR can prevent the persistence reordering bug we found (Bug 1) but this bug is still possible if the compiler reorders instructions. Most bugs found by Silhouette are logic bugs and they are unaffected by eADR.

The future of PM: Intel® discontinued the Optane™ PM product line [19] in 2022, leading to uncertainties about the future of PM. Fortunately, emerging Compute Express Link™ (CXL™) aims to provide a cache-coherent interconnect for processors, memory expansion, and accelerators, and supports expanding PCIe® devices as storage-class memory [9]. However, with storage-class memory, stores may be held in Processor/CXL Device caches or Memory Device Write buffers for performance reasons [37], leading to potential persistence bugs. Furthermore, CXL memory pooling allows multiple hosts to share memory in different failure domains, raising crash-consistency issues even for volatile memory [10]. We expect that Silhouette's approach will be applicable for detecting bugs in such systems.

6 Conclusions

This paper presents Silhouette, a new testing framework for finding crash-consistency bugs in PM-based file systems. Silhouette uses invariants to check whether PM file systems correctly implement their crash-consistency mechanisms, such as journaling and replication. If these checks pass, then all stores associated with the consistency mechanism are considered protected, and only the unprotected stores are checked. Silhouette also uses a simple, novel exploration strategy that scales well with the number of in-flight stores. Together, these strategies enable Silhouette to scale testing and find over a dozen new bugs in PM file systems.

7 Acknowledgments

We thank our shepherd Ramnatthan Alagappan and anonymous reviewers for their invaluable feedback. This work is sponsored by NSF DGE-2146354 and FSU. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or FSU.

A Artifact Appendix

Abstract

This artifact includes the Silhouette prototype, configurations, test cases, and scripts for reproducing key findings from Silhouette, such as identified bugs and the number of crash plans. Additionally, it provides a virtual machine image (in QCow2 format) with a pre-configured environment, including a compiled Linux kernel supporting the tested PM file systems.

Scope

The artifact supports the following use cases:

- Classifying PM stores into mechanism-protected and unprotected stores, as shown in Figure 1.
- Reproducing the bug detection process for the newly discovered bugs listed in Table 2.
- Testing various crash plan generation schemes and generating results similar to Table 3.
- Evaluating Silhouette with different numbers of VMs and workloads to achieve runtime results comparable to those in Section 5.3.

Users can also extend the artifact to test additional file systems, detect untested mechanisms, or implement custom crash plan generators with modifications.

Contents

The artifact package includes the following components:

- `README.md`: Instructions for using and evaluating Silhouette.
- `codebase/`: The source code of Silhouette, including the LLVM instrumentation component, scripts for execution, workloads, and related files.
- `evaluation/`: One-click scripts for reproducing bug findings and evaluating Silhouette's scalability. Each subdirectory contains a `README` file with explanations and guidance for reproduction and understanding test outputs.
- `thirdPart/`: Source code for the tested file systems.

Hosting

Silhouette is open-sourced at `https://github.com/iaoing/Silhouette` with the artifact hosted in the `fast25_artifact` branch. For convenience, the artifact is also hosted on Chameleon Cloud [4], an NSF-funded testbed for computer science experimentation: `https://www.chameleoncloud.org/experiment/share/3c807f1d-80db-443c-8d88-c645fa3695e8`. Users may choose to test Silhouette either on their local machines or using Chameleon Cloud. The pre-configured VM image is available on Zenodo: `https://zenodo.org/records/14550794`.

Requirements

Testing Silhouette on Chameleon Cloud requires only a web browser. Users without an account or allocated resources may need to apply for a day pass to use Chameleon Cloud.

For testing on a personal machine, the following requirements apply:

- A bare-metal machine running Linux (we have tested on Ubuntu-22) or a Linux virtual machine supporting nested virtualization.
- `QEMU` installed, with support for emulating NVDIMM and Intel Xeon Scalable processors (2nd generation or later).
- `Memcached` and `Python3.10` installed.
- Additional dependencies (e.g., `automake`) and Python packages can be installed using the `install_dep.sh` script in the repository.

Other components, such as LLVM, the LLVM-compiled Linux kernel, and related dependencies, are included in the provided VM image, allowing users to test Silhouette without affecting their host environment.

References

[1] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[2] 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, 2016.

[3] Miao Cai, Junru Shen, Bin Tang, Hao Huang, and Baoliu Ye. FlatFS: Flatten hierarchical file system namespace on non-volatile memories. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 899–914, 2022.

[4] ChameleonCloud. Chameleoncloud. https://chameleoncloud.org/, 2025.

[5] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[6] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: a persistent b+-tree with low tail latency. *Proceedings of the VLDB Endowment*, 13(12):2634–2648, 2020.

[7] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 799–812, 2020.

[8] 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 *Proc. of the Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009.

[9] CXL Consortium. Compute Express Link: The breakthrough cpu-to-device interconnect CXL. https://www.computeexpresslink.org/, 2025.

[10] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. Persistent memory research in the post-optane era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems*, pages 23–30, 2023.

[11] 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*, pages 503–516, 2021.

[12] 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)*, 2014.

[13] Daniel Fryer, Mike Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the integrity of transactional mechanisms. *ACM Trans. Storage*, 10(4), oct 2014.

[14] 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*, pages 100–115, 2021.

[15] Xinwei Fu, Dongyoon Lee, and Changwoo Min. DURINN: Adversarial memory and thread interleaving for detecting durable linearizability bugs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 195–211, 2022.

[16] 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*, pages 415–428, 2021.

[17] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, 2018.

[18] Intel. eADR: New opportunities for persistent memory applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[19] Intel. Intel Q2 2022 earnings. https://d1io3yog0oux5.cloudfront.net/_7476eb634cf21033bf2ce4974e02203e/intel/db/887/8856/prepared_remarks/Intel-CEO-CFO-2Q22-earnings-statements-1.pdf, 2022.

[20] Intel. Intel 64 and IA-32 architectures software developer's manual (combined volumes). https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4, 2023.

[21] 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*, pages 804–818, 2021.

[22] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[23] 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, 2022.

[24] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, 2022.

[25] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A high performance persistent range index using PAC guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 424–439, 2021.

[26] R Krishnakumar. Kernel korner: kprobes-a kernel debugger. *Linux Journal*, 2005(133):11, 2005.

[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 (USENIXATC 14)*, pages 433–438, 2014.

[28] Hayley LeBlanc, Shankara Pailoor, Om Saran KRE, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 1–20, 2023.

[29] Hayley LeBlanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. SquirrelFS: using the rust compiler to check file-system crash consistency. *arXiv preprint arXiv:2406.09649*, 2024.

[30] 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*, pages 462–477, 2019.

[31] 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*, pages 1187–1202, 2020.

[32] 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*, pages 411–425, 2019.

[33] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020.

[34] 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. *ACM Trans. Database Syst.*, 17(1):94–162, mar 1992.

[35] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Crashmonkey and ACE: Systematically testing file-system crash consistency. *ACM Transactions on Storage (TOS)*, 15(2):1–34, 2019.

[36] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *FAST*, volume 19, pages 31–44, 2019.

[37] Mahesh Natu and Thomas Won Ha Choi. Compute Express Link (CXL): Supporting persistent memory. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL-2.0-Presentation-Persistent-Memory-20210615_FINAL.pdf, 2023.

[38] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386, 2016.

[39] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276, 2014.

[40] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.

[41] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[42] Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. SEPH: Scalable, efficient, and predictable hashing on persistent memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 479–495, 2023.

[43] Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, and Jiwu Shu. Revisiting secondary indexing in LSM-based storage systems with persistent memory. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 817–832, 2023.

[44] Zhonghua Wang, Chen Ding, Fengguang Song, Kai Lu, Jiguang Wan, Zhihu Tan, Changsheng Xie, and Guokuan Li. WIPE: a write-optimized learned index for persistent memory. *ACM Transactions on Architecture and Code Optimization*, 21(2):1–25, 2024.

[45] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[46] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[47] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. MadFS: Per-file virtualization for userspace persistent memory filesystems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 265–280, 2023.

[48] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.