Karlsruhe Institute of Technology

Crash Consistency Testing for Cross-Media File Systems using Persistent Memory and NVMe

Bachelor's Thesis
submitted by

Lucas Wäldele

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Lukas Werling, M.Sc.

22. April 2023 – 22. September 2023

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, September 22, 2023

Abstract

Verifying crash consistency guarantees for file systems is hard because of the multitude of software and hardware layers involved. Vinter is a software to test file systems backed by persistent memory (PM). Revin adapts this approach to file systems backed by SSDs accessed through NVMe. However, there are cross-device file systems using both devices in tandem, because PM offers fast persistent writes while SSDs offer great storage capacity. To make claims about crash consistency for these systems, we need to analyze both devices at the same time.

For this purpose we present Permanent, a cross-media tester built on Vinter and Revin. It is able to test full systems for crash consistency guarantees. Permanent traces commands to both PM and NVMe storage devices in a virtualized environment. From this trace, Permanent generates crash images for both devices, taking hardware effects like command reordering and caching into account. From the generated images, Permanent extracts the semantic file system state. Based on the amount of distinct semantic states, it automatically deduces whether crash consistency guarantees like atomicity are satisfied.

To evaluate Permanent, we test ZIL-PMEM, a cross-media extension of ZFS. We find that Permanent is able to produce expected semantic states of the file system, and that ZIL-PMEM can recover from crashes in many different scenarios.

Contents

Chapter 1

Introduction

File system consistency is important to prevent loss of data, metadata inconsistencies, and file system corruption. However, file systems are used on top of storage hardware which is unreliable in the event of a crash. Crashes occurring during execution can lead to previously issued writes not being persisted, only partially persisted, or persisted in an unexpected order. To make a file system robust, it needs to be aware of how the underlying hardware devices function, and how they may behave in the case of a crash.

It is difficult to definitively prove crash consistency guarantees because of the complexity of the underlying hardware and software layers. A simpler approach is to search for consistency violations.

We look at two tools which can automatically detect breaches in consistency guarantees. Vinter [13] does this for file systems backed by byte-addressable persistent memory, like Intel Optane memory [1]. Vinter runs test cases in a virtualized environment to trace the executed instructions. From this trace it generates a number of crash images that are possible in a real system, being aware of instruction reordering and caching on the CPU [13]. Vinter applies a heuristic to find the most relevant crash images and reduce computation time [13]. The resulting crash images are mounted and Vinter extracts their semantic file system states, grouping crash images representing the same semantic state together. The number of possible states is used to automatically test guarantees like atomicity [13]. Due to the virtualization approach, Vinter is capable of testing full systems without source code modification or recompilation [13].

The second tool, Revin [16], applies this approach to block-based file systems using the NVMe protocol. Like Vinter, it takes NVMe write caching and write reordering effects into account when creating crash images, emulating a real hardware device [16].

There are file systems like ZIL-PMEM [17], Strata [14], or device setups like DPWC [11], which use both PM and NVMe storage at the same time. We cannot

test these file systems with Vinter or Revin, because both devices are accessed in different ways, and they differ in behaviour in the case of a crash. Because Revin is strictly block-based, it cannot handle PM memory accesses. Vinter can only test a cross-media file system by using a PM device in place of the NVMe device, thereby ignoring NVMe-specific hardware behaviour. Additionally, simulating two devices at the same time poses new problems which are not present in a single-device simulation, like the possibility of command reordering between different devices.

In this work we combine Vinter and Revin into such a unified crash-consistency tester: Permanent, the persistent memory and NVMe non-deterministic tester. This tester is able to analyze cross-media file systems that use PM and block-based storage through NVMe in tandem.

We discuss the implications on Permanent of having two devices with different semantics, and redesign the parts of Vinter's and Revin's pipeline to simulate both hardware devices and the interaction between them in a way that is representative of real hardware behaviour. Permanent inherits Vinter's unique features like unmodified full system testing and automatic crash consistency checking.

Based on our design we implement a working prototype of Permanent. We run tests on ZFS with ZIL-PMEM [17] using our prototype. We find that ZIL-PMEM handles all test cases implemented by Vinter gracefully.

Chapter 2

Related Work

In this chapter we explain the terms and concepts integral to this work. We first give a short introduction to different types of persistent storage. We describe the concept of crash consistency in the context of file systems, which is the focus of our work. Next we introduce virtualization to explain how it can be leveraged to test full systems. Then, we describe the software that we build upon in detail.

2.1 Persistent Storage

To store data permanently, one requires hardware that can retain information after being disconnected from a power source. There are different types of hardware that support such persistent storage.

2.1.1 Block-Based Storage

Some persistent storage is accessed in block-granularity. These blocks usually have a size of 512 to 4096 bytes; only entire blocks are written or read, and there are no sub-block operations [15]. Writing to a single block is often atomic, which is important for crash consistency considerations [15].

NVME

Non-Volatile Memory Express (NVMe) is a protocol used to access block-based storage, like SSDs [15]. Commands are sent in submission queues, and the device signals completion in completion queues. Commands include reading data, writing data, writing zeroes, and flushing, among others [15]. Commands can be freely reordered by the device, and writes may be cached on the device without making them persistent. To persist data, the write cache needs to be flushed with a global flush command [15].

2.1.2 Persistent Memory

Other devices, like Intel Optane Persistent Memory [1], can be accessed with a smaller granularity: individual bytes are addressable. We refer to these devices as persistent memory (PM) devices. They are usually accessed with standard memory load and store instructions by mapping the PM device into the virtual address space [1]. Data is only persisted when bypassing the cache with a non-temporal store, or when the respective cache lines are written back. On x86 this can be enforced with special instructions like `clflush`, `clflushopt`, and `clwb` [12]. These instructions may be reordered as they are executed; ordering these instructions relative to each other is achieved by a memory fence (`sfence` or `mfence`) [12].

2.2 File Systems

File systems are an abstraction on storage mediums. Instead of interacting with the devices directly, users operate on files, which are collections of data with variable length [10]. File systems can be backed by block-based storage or PM, or a combination of the two. We refer to a file system that uses both technologies as a cross-media file system, or hybrid file system.

2.2.1 Crash Consistency

Persistent storage devices are vulnerable to crashes, for example in the case of a power outage. In this case, the device may still carry data in volatile caches, or be in the middle of a larger operation. Data in the volatile caches may be lost, while operations may finish execution or be interrupted (e.g., a torn write) [10]. To prevent data loss, it is therefore important for the file system to be aware of the crash semantics of the underlying hardware device. For example, writes to SSDs are often atomic with respect to a single block, while writes to PM memory are only persisted at the granularity of cache lines (usually 64 bytes) [1, 15]. In this case, crash consistency means that no matter when a crash occurs and what operations are completed or interrupted, the file system will remain in a consistent state, or in one of a set of consistent states that are well defined by the file system [10].

2.2.2 ZIL-PMEM

ZIL-PMEM [17] is based on the ZFS file system. ZFS is a file system where file system operations (e.g., creating or deleting a file) are carried out through

atomic transactions instead of individual block writes [9]. Transactions may be grouped together. Before they are committed, they are stored in the ZFS intent log (ZIL) [9]. This log can be stored on disk, or, in the case of ZIL-PMEM, on persistent memory. ZIL-PMEM tailors the ZIL to the specific performance characteristics of PM [17].

2.3 Virtualization

Instead of running on actual hardware, programs can be run in an environment that emulates said hardware, being transparent to the program running on top of it. The piece of software performing this emulation is called the hypervisor, or virtual machine monitor [10]. Virtualization offers the possibility to run software without having the necessary hardware devices physically present. In our case this could be an SSD, or a PM module.

2.3.1 QEMU

QEMU is an emulator which can create virtual instances of entire machines (called the guest), including various CPU architectures and I/O devices [4]. It can emulate hardware with dynamic binary translation, or be used in conjunction with a hypervisor like KVM to use the host architecture directly [4].

TCG

QEMU uses its Tiny Code Generator (TCG) to translate guest instructions into an intermediate representation which is independent of the guest and host architecture [4]. TCG instructions are collected into basic blocks for increased performance. These TCG basic blocks are then translated to native host basic blocks to be executed directly on host hardware [4].

TCG plugins allow adding custom functionality to a QEMU quest execution. Individual guest instructions can be inspected as they are translated, and a callback may be registered to be notified when this specific instruction is executed [4]. Custom data may be passed to the callback to identify individual instructions [4].

2.3.2 PANDA

PANDA is a framework for dynamic analysis built on top of QEMU [2]. Whole system executions can be recorded and replayed. It employs a plugin system to allow for different analysis functionality to be combined [2].

2.4 Vinter

Vinter [13] is a software to analyze full systems running a file system on PM. Vinter traces accesses to PM and generates possible crash states resulting from that trace. It extracts the semantic state of these crash states and can automatically test consistency guarantees based on the possible intermediate states during the operation [13]. It is split into different parts which are explained below.

2.4.1 Tracer

Vinter uses PANDA to trace PM accesses. A previously specified file system operation is executed in a virtualized environment. It records load, store, cache flush/write back, and fence instructions that affect the persistent memory, along with hypercall instructions to separate different logical operations [13]. All of these are recorded in order and stored for the following stages of the analysis.

2.4.2 Crash Image Generator

The traces are fed into the crash image generator. Its task is to generate images of the PM at different points in the trace [13]. Instead of assuming sequential execution of the traced instructions, the crash image generator creates crash images that are possible on a real system. More specifically, it is aware of the effects of instruction reordering and caching: memory operations may be executed in a different order than recorded in the trace, and data may still be in the CPU caches or already written back to the device [13]. To implement this, the crash image generator examines instructions between each two ordering points and generates possible subsets of them. It creates a PM image by replaying all store instructions that are guaranteed persisted, i.e., writes that have been explicitly flushed (including fencing after the flush), and then overlaying the chosen subset of unpersisted writes [13].

Vinter uses a heuristic to limit the amount of crash images generated. First, it creates a crash image with all stores persisted. It records all reads that occur while walking the directory structure recursively and requesting file data and metadata. The resulting trace is called recovery trace [13]. Then, when generating crash images, it only regards stores going to PM addresses that appear in the recovery trace. The justification for this is the following: if traversal and recovery code does not read from an address, it is unlikely that the data at this address will have an impact on the semantic state of the file system [13]. Experiments have shown that for the file systems and test cases used in Vinter, this is a valid consideration [13].

2.4.3 Tester

PM images generated by the Crash Image Generator are fed into the tester, which extracts the semantic state. Vinter mounts the image in a virtual environment and dumps the file system content by walking through the directory structure, extracting metadata for each file system entity and the content of each file [13]. Two different PM images may resolve to the same semantic state. The tester records all different semantic states that may appear, as well as the set of crash images that represent this state. From these it derives crash consistency guarantees: an operation is faulty if there are possible images that represent an invalid semantic state. An operation is single final state (SFS) if all crash images that are possible exactly at the end of the operation resolve to a single semantic state [13]. An operation is atomic if all crash images that are possible between the beginning and the end of the operation resolve to one of exactly two semantic states: the before state and the after state [13]. Both of these states must be valid.

2.5 Revin

Revin [16] is based on Vinter and analyzes block-based file systems using NVMe. It consists of the same parts as Vinter, with the following differences:

2.5.1 Tracer

Revin traces NVMe commands in a virtualized environment, specifically read, write, and flush [16]. Unlike Vinter, this is accomplished with QEMU's NVMe emulation capabilities with a few modifications [16]. Revin uses QEMU's trace event feature to generate trace entries. Instead of using hypercalls to signal reached checkpoints, Revin uses QEMU's serial port [16].

2.5.2 Crash Image Generator

Similar to Vinter, Revin's crash image generator also considers command reordering, as well as on-device caching, but in this case in the context of NVMe: writes to different blocks may be reordered, and unless a write command that is marked as completed is flushed, it may still be unpersisted [15, 16]. Combining these two effects leads to the following algorithm: Revin generates permutations of operations between each two flushes (called a transaction [16]). Sequential parts of these permutations are then applied to the guaranteed persisted content of the NVMe device. To limit the amount of generated crash images, Revin only gen-

erates a fixed number of random permutations; this is called the random algorithm [16].

The Vinter-heuristic based on recovery-code reads can also be applied to Revin and be used in conjunction with the random algorithm; however, it was deemed not very effective and was thus not implemented completely [16].

2.5.3 Tester

The semantic state extraction works like Vinter: Revin runs extraction code inside a virtualized environment, and records the different semantic states that may appear. Automatic testing of SFS and atomicity guarantees are also possible [16].

For stronger metadata consistency guarantees, Revin checks that all inodes have matching `.` (current directory) and `..` (parent directory) entries. The inode of a file must also correspond to the inode found by walking through the directory tree with `lstat`.

Chapter 3

Design

In this chapter, we describe the considerations and design decisions we made to combine Vinter and Revin into Permanent. First, we give an overview of the goals we want to achieve with our design. Then, we describe the complete testing pipeline to illustrate how the different parts of the Permanent fit together. We discuss general considerations we made when combining Vinter and Revin by defining a device model for PM and NVMe and extending it to a hybrid device model. After this, we describe the design of the individual parts in the pipeline. Like in Vinter and Revin there are three parts: the tracer, the crash image generator, and the tester.

3.1 Goals

The main goals of our work are as follows:

- It is important that the original core functionalty remains accessible, i.e., Permanent is able to handle PM-exclusive file systems, and NVMe-exclusive file systems also. In these two cases, we want to be able to generate the same trace, the same set of crash images, and the same semantic state results as Vinter or Revin did before.
- We also want to apply the core features of Vinter and Revin to hybrid file systems. This includes the ability to test full systems without source code modification or recompilation, and the testing of single final state and atomicity guarantees as described in Section 2.4.3.
- Permanent should create hybrid crash images that are representative of the expected hardware behaviour. This includes the behaviour of command reordering and caching, but also possible ordering constraints between com-

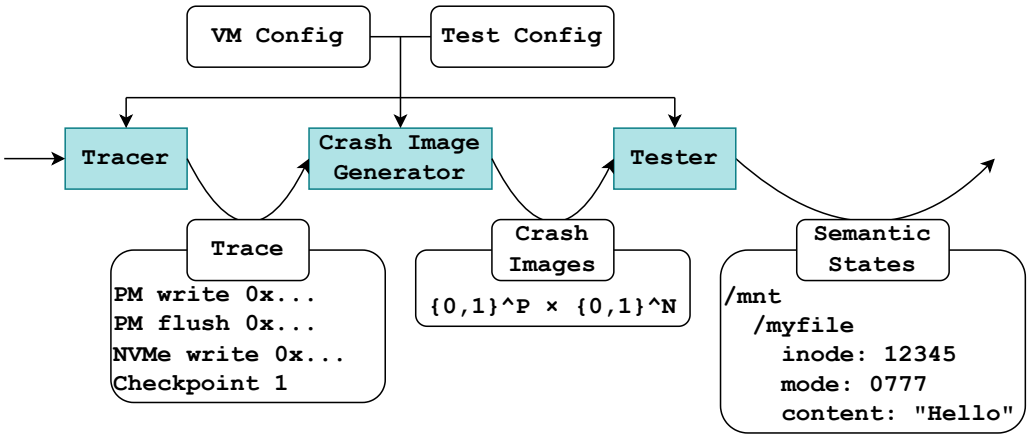

Figure 3.1: Permanent's pipeline. The three main components are the tracer, the crash image generator, and the tester. Arrows show the data flow between pipeline stages. For the crash images, P is the size of the PM device and N is the size of the NVMe device.

mands to different devices. This is a new consideration for our combined program, as Vinter and Revin each only support a single device.

- Lastly, we want to retain performance close to Vinter.

3.2 Complete Pipeline

We describe an execution of the complete pipeline, shown in Figure 3.1. The pipeline design is directly inherited from Vinter and Revin.

The sequence of events for a full Vinter execution is as follows:

1. The user specifies a VM configuration file and a test configuration file. These are put in an empty directory along with a PM or a NVMe base image or both, depending on the file system type. The file specifications are adapted from Vinter. Each VM configuration file contains:

   - file system type (PM only, NVMe only, or hybrid),
   - physical range of PM addresses (for PM and hybrid systems), and
   - information to start VMs: kernel path and initramfs path, as well as configuration options for the virtualization.

   Additionally it contains shell commands to execute inside the VM, which are the same for every test case. There are three commands:

   - a trace command prefix for file system setup,

- a recovery command for the Vinter heuristic, and
- a dump command prefix to extract the semantic state of the file system.

Each test configuration file contains:

- the trace command suffix, which appended to the trace command prefix forms the complete test case,
- a range for valid checkpoint values, and
- a dump command suffix with additional file system operations to check for file system corruption after all data has been extracted.

2. The tracer is executed once with the trace command, tracing all relevant device accesses for the specified file system type, as well as checkpoints. It produces the trace for crash image generation.

3. The crash image generator moves through the trace and generates crash images at relevant points. It creates a set of crash images, as well as metadata describing which crash image was generated at which point in the trace, and to which device it belongs.

4. The tester receives a set of hybrid crash imagse. For each of them, the tracer is started with the dump command to extract the semantic state. A file system is corrupted (semantic state $\perp$) if it cannot be traversed, or if executing the dump command suffix produces an error. Otherwise we compare the semantic state to previously encountered states. For each semantic state, the tester remembers which crash images led to it.

3.3 Device Model

We regard a simplified model which we take as a base for our device simulations, and which will also serve as a base to reason about the completeness of our approach. This model takes command reordering and volatile caching into account.

Each device has access to an addressable area of persistent storage comprised of storage regions. The device has three types of commands that may be sent to it: write, flush, and ordering commands. The device may reorder commands without shared dependencies (in this case, writes and flushes that do not affect the same storage region on the device).

Writes are comprised of a target address and data. Write data is stored in a volatile cache once the write command is executed. A write in the volatile cache may be persisted at any point, but it might also stay volatile.

A flush command can explicitly persist data. Once it is executed, the content of the volatile cache referring to same storage region as the flush will be persisted.

To serialize other commands there exists a special ordering command. Commands may not be reordered beyond these ordering points. After an ordering point, all commands that came before have been executed. Note that for a write this only means the data now resides in the volatile cache, not that it has necessarily been persisted.

Based on these rules, each device has a set of valid states it may be in at any point in time, depending on the order of executed operations. We call the content of the persistent area of the device persisted content, and the content of the volatile caches unpersisted content.

3.3.1 Application to PM

This simplified model maps closely to a PM device. Here the volatile cache is the hardware cache of the x86 CPU; storage regions are the individual cache lines. Writes are standard x86 load and store instructions; flushes are x86's `clflush`, `clflushopt`, and `clwb`. Writes and flushes to different cache lines may be reordered among each other, except `clflush` which is ordered with respect to itself and other writes [12]. Ordering commands are the `sfence` and `mfence` instructions.

We adapt the model somewhat to better approximate real x86 hardware: unaligned writes are split on eight byte boundaries. Additionally, x86 supports non-temporal writes which bypass the cache and are persisted on execution [12].

This is the same model Vinter uses for PM crash image generation.

3.3.2 Application to NVMe

Our model also maps to real NVMe hardware. Writes are cached in a volatile write cache on the device, and storage regions are the atomic blocks [15]. One exception is that flush and ordering commands are merged into one. With an NVMe flush command, one can force the write cache to be flushed. All commands that came before are now executed and all writes persisted. An NVMe flush thus acts as a global flush command, followed by an ordering instruction.

Strictly speaking, this is only true for write commands which have been marked as completed before the flush (see discussion in [16]), but in the following we will treat a flush as totally ordering. We will also discount the possibility of having more than one command queue for an NVMe device.

Similar to our PM model, we split NVMe writes on atomic block boundaries. We assume an atomic block size of 512 bytes. There cannot be unaligned writes in NVMe.

With the exception of write splitting, this is the model Revin uses for NVMe crash image generation.

3.3.3 Write Reordering

It is important to note that there are two times writes may be reordered: once as commands when they are transferred to the volatile caches, and once when the cache content is persisted. Even writes that were initially ordered may have their content persisted in a different order.

This has the following implication: the set of possible states of the persisted content between two ordering points never depends on the order of the writes. They can still be persisted in any order, no matter the order in which they were executed. This means we can treat all writes as immediately executed in the order they appear in the trace without losing information. We cannot do the same for PM flushes, however. Flushes can restrict the reordering of writes being persisted.

3.4 Hybrid Device Model

Our goal is combining the tracing and crash image generation of PM and NVMe devices in such a way that we can reconstruct an accurate state of the devices without losing relevant information. This includes information about how operations on different devices relate to each other.

3.4.1 Cross-Device Effects

The first thing we need to consider is whether there are visible effects a command of one device can have on the state of the other device. For example, commands on one device could affect the persistent storage or the caches of another. This is not the case on an x86 Linux system like the ones we test. PM writes, flushes or fences do not change data on the NVMe device or the state of the write cache. Similarly, executing and NVMe command will not change the content of the PM device, and won't force data out of the processor caches. Of course, cache lines may be flushed in the meantime, but not as a necessary consequence of executing an NVMe command. So in our extended device model, we assert that commands to one device do not affect the other in any observable way. This excludes ordering constraints, described in the following section.

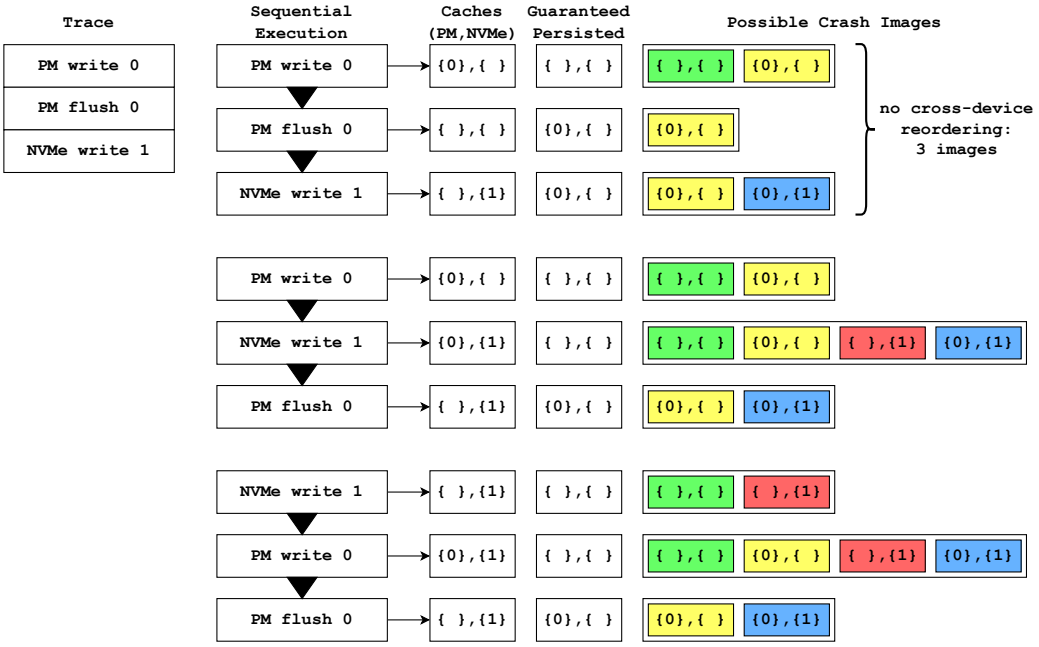

Figure 3.2: Impact of cross-device reordering. The trace is shown on the left. To find all crash images, we look at all possible sequential executions of the trace. Without cross-device reordering, there is only one possible sequential execution, because PM writes and PM flushes are ordered, and the NVMe write has to be executed after the PM flush. After every instruction we inspect the current state of the volatile caches and the guaranteed persisted content to find crash images at this point. Without cross-device reordering there are three possible crash images. With cross-device reordering there are now two more possible sequential executions, which combined lead to four possible crash images. Thus, cross-device reordering allows for a bigger set of crash images.

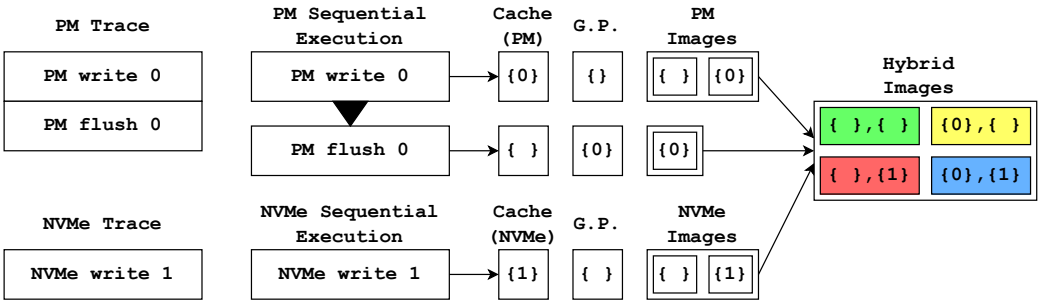

Figure 3.3: Crash image generation with free cross-device reordering by tracing and generating crash images separately. We generate single-device crash images from the separate traces. For the PM trace and the NVMe trace there is only one possible sequential execution, which is depicted here. Hybrid images are all possible combinations of PM and NVMe crash images. The trace is taken from Figure 3.2. We observe that the separate algorithm generates the union of the three sets of crash images possible with the combined sequential executions.

3.4.2 Reordering Between Different Devices

Based on our device model, we established how command reordering on a device can increase the set of possible states the device may be in at any given time. A new consideration when reasoning about hybrid operation is how commands on different devices may be reordered with respect to each other.

We lose the advantage of having global ordering points. In Vinter, at a memory fence we could be certain that all operations which came before in the trace will also be executed before. However, our previous device model restricts ordering of a device command only beyond ordering points of the same device. It makes no statement about how this command may be ordered in respect to commands on another device. There are several ways in which we could extend our model to define cross-device ordering constraints.

No Cross-Device Reordering

The first version prohibits reordering around operations to a different device. An operation on device 2 can only be executed after all previous commands on device 1 have been executed, and vice versa. This means that each pair of operations to different devices forms an implicit ordering point. Using this approach restricts the amount of legal crash images significantly compared to free cross-device reordering (Figure 3.2).

Free Cross-Device Reordering

The second version imposes no ordering restrictions between commands on different devices at all. Even ordering points of different devices could be reordered. Using the Vinter crash image generation algorithm which only generates images from preceding commands in the trace would not be applicable, because it is undefined which commands from the other device came before.

Since there are no timing or ordering constraints between these devices at all, there is a simple approach to tracing and crash image generation that follows this extended model: tracing PM and NVMe operations separately, generating crash images from these traces independent of each other, and later combining each PM image with each NVMe image. This approach results in $P \cdot N$ hybrid images for P PM images and N NVMe images (Figure 3.3). Note that this relies on the assertion we made in Section 3.4.1 about the absence of other cross-device effects.

Resulting Model in Vinter

We now consider which model best describes the hardware devices we use.

Of course, there are more possibilities like the two extremes we mentioned above, like reordering being limited to a specific range around each command. However, we argue that our situation is best described by no cross-device reordering, for the following reasons:

With the Linux kernels we used, each NVMe operation is surrounded by two memory fences on kernel entry, and three memory fences on kernel exit. This means all PM operations occurring before an NVMe command are also executed before, as long as the PM fences themselves cannot be delayed beyond the NVMe command. As each NVMe command submission requires memory accesses, we assert this to be the case. The reordering presented in Figure 3.2 is thus impossible in our use case.

The opposite is harder to reason about: NVMe commands are put in a submission queue, but there might be a delay between command submission and command execution on the device [15]. Before their execution, there might already be other PM instructions being executed.

For NVMe writes, this asynchronicity can be safely ignored, following the remark in Section 3.3.3. For crash image generation there is no difference between a write command that is in the submission queue, and a write command that has already been executed with data now residing in the write cache. In both cases the write data will be persisted some time in the future. Thus we treat every NVMe write as immediately executed, without having to observe the details between submission and completion.

The same is not true for NVMe flushes. However, we also treat them as immediately executed, because like Revin we have no information about the completion of NVMe events.

It follows that all PM operations occurring after an NVMe command are executed after and vice versa, so with some simplifications the extended model without cross-device reordering is accurate to our hardware.

This approach requires ordering information between operations on different devices. It is not sufficient to have two separate traces. This has a fundamental impact on our tracer design, described below.

3.5 Tracer

The Tracer uses a virtualized system to trace interactions with the file system through the respective operations. Vinter already traces PM accesses, including reads, writes, cache flushes, and fences; Revin traces NVMe reads, writes, and flushes. All trace entries carry information with them which is relevant to the generation of crash images. The index of a trace entry in the trace is called its trace ID.

Writes carry destination address, length and data; PM writes can optionally be non-temporal, meaning they will be persisted to the device immediately. This mechanism does not exist for NVMe devics.

Reads carry destination address and length. They do not need to carry data, because we are only interested in where the reads occur for post-success tracing, not what data they carry. Should it be required, the data on the device can be reconstructed by applying all previous writes to the device image.

PM fences and NVMe flushes carry no data because they are global, while PM flushes carry the destination address and are distinguished by their instruction type, because different x86 flush instructions have different semantics and ordering guarantees [12].

We also add checkpoints to our trace to separate logical operations, and to specify where the setup ends and the actual file system operation begins. Each checkpoint carries a number with it that can be used later to determine at which logical operation a crash image originated. Checkpoint numbers are always contiguous.

As described above, we want one single trace where PM and NVMe operations are recorded in the order they appear, instead of two separate ones. This could be accomplished by tracing PM and NVMe operations separately, attaching a timestamp to each trace entry and combining them later. However, we decided to unify Vinter and Revin tracing to form a single combined trace during execution. This makes timing and joining obsolete, increasing overall performance and simplifying the tracer design.

There are three places in Vinter where the tracer is utilized, and it is the same for Permanent.

- Analysis tracing: we execute the trace command from the configuration files and record all writes, fences and flushes.

- Post-success tracing: this is used for Vinter's heuristic. When we generate crash images, we apply all stores to form a fully persisted image. We execute the recovery code on it and record the addresses of all reads. Then, in crash image generation, we can use these reads to limit randomization to those regions where the recovery code read from.

- Post-failure tracing: in the tester, we run state extraction code on the generated crash images. This state extraction is also done in a virtualized environment.

3.6 Crash Image Generator

The crash image generator is now in charge of simulating two devices, which means each crash image for our hybrid file system consists of two separate images. Some considerations made in Vinter and Revin apply here, while others are new for hybrid systems.

3.6.1 Replay Algorithm

The conceptually simplest approach to finding all crash images for a given trace is the one we used in Figure 3.2: we consider all possible sequential executions of the trace, and at each point inspect caches and persisted content to find crash images. However, the amount of sequential executions grows exponentially in the worst case. Therefore we use a different approach: we move through the trace once sequentially, remembering which commands are ordered with respect to each other, and which ones are free to be reordered. We only generate crash images when we encounter an entry in the trace which, by being executed, would remove elements from the set of possible crash images. We call these points crash image generation points. We first illustrate how this is accomplished for NVMe-only and PM-only systems, and then move on to our hybrid case.

NVMe

NVMe writes can be reordered among each other. As long as there are only writes in the trace, we do not have to create crash images. Consider three NVMe writes. When we arrive at a fourth one, we can still generate all crash images that were possible before the fourth write by simply treating it as not persisted. Therefore do not lose information at this point when we don't generate crash images. Only when an NVMe flush arrives does the set of possible crash images shrink to one, so we have to generate crash images before this flush is executed. With this algorithm we find all NVMe crash images which would have been possible with an arbitrary sequential execution of the trace.

This replay algorithm is implemented by Revin, with the exception that Revin splits the full trace at NVMe flushes instead of replaying NVMe commands sequentially [16]. We adapt this approach for Permanent.

PM

It is similar for PM, although complexity increases with the separation of flushes and fences. As long as we only have writes, the set of possible crash images grows. The same is true when we encounter flushes: they do not change the set of

possible crash images, because they are not immediately executed. Only when we have a fence with at least one flush preceding do we need to generate crash images, because then the set of possible crash images shrinks, as the cache content referred to by the flush can no longer be treated as unpersisted. Note that on a fence with no preceding flushes, we do not have to generate crash images. The execution of previous writes may be ordered then, but they may still be persisted in any order (see Section 3.3.3).

This replay algorithm is very close to Vinter's, although it leads to less crash image generator invocations because Vinter always generates crash images on a fence with preceding writes [8]. However, the difference is insignificant, as this is a special case which has no useful application. Programs working on PM always use fences in combination with flushes, because without flushes the order of writes being persistent is still arbitrary.

Hybrid

Again we move through the trace sequentially, remembering instrucions and their ordering guarantees among each other. The replay algorithms stay the same for the single devices, which is why we reruse them from Revin and Vinter. Additionally we now look at points in the trace where we have two adjacent commands to different devices. We call these points transition points.

In Section 3.4.2 we established that we should prohibit reordering of commands to different devices. However, this does not mean that every transition point is also a crash image generation point. Consider a PM write followed by a PM fence followed by an NVMe write. The PM write is guaranteed to be executed before the NVMe write, but they may still both be in the respective volatile caches, meaning that the order of them being persisted is not defined at this point. The set of possible crash images grows. Again, the same consideration applies: Only when there have been PM flushes do we have to generate crash images at transition points. Note that in practice this never happens, as every NVMe operation is surrounded by PM fences.

Taking this consideration into account, our behaviour for hybrid crash image generation is the same as in the single-device case:

- When a PM fence occurs, we have to generate crash images only when there was a PM flush since the last fence. The current state of the NVMe device does not influence this decision.

- When an NVMe flush occurs, we only have to generate crash images when there are NVMe writes in the write cache. This decision is also independent of the PM device.

We also generate crash images when we encounter a checkpoint, since this is needed for single final state testing. The checkpoints impose no further ordering restrictions.

Here, generating crash images refers to hybrid crash images. It is not sufficient to generate them only for the device where the ordering point originated.

3.6.2 Generating Hybrid Crash Images

Now that we established when hybrid crash images are generated, we specify how they are generated.

Since both devices in a hybrid file system setup are distinct entities, we can generate crash images for them separately, without consideration for the state of the other device. This makes it possible to reuse Vinter's and Revin's crash image generation algorithms.

The novel consideration for Permanent is which individual crash images can then be combined to form a hybrid crash image. As we established in Section 3.4.2, in the general case Permanent is more restrictive than combining every PM image with every NVMe image. We use the following approach: at each point where crash images are generated, Permanent creates an index which contains the set of PM crash images and the set of NVMe crash images generated at that point. All hybrid crash images at this point are described by the cartesian product of these two sets. In the general case, the amount of hybrid crash images grows quadratically with the amount of single-device crash images.

There is one trivial optimization we implemented for Permanent's hybrid algorithm: If the set of possible crash images for a single device has not changed since the last time crash images were generated for it, we simply reuse the last set of generated crash images for this device. This improves efficiency: when there are many crash image generation points triggered by one device while the other does not receive any writes, we do not spend unnecessary time generating crash images from the same writes multiple times. It should be noted that we very likely end up with different crash images when this optimization is disabled. Without it, we would generate new random crash images. However, since the set of possible crash images is the same, the correctness of our approach does not change.

3.6.3 Limiting the Amount of Crash Images

Because the crash image space can grow exponentially with each trace entry, we do not generate all crash images exhaustively. Instead, we use a random subset of possible crash images. There has been discussion on the effectiveness of various heuristics and random algorithms for both devices in Vinter and Revin [13, 16]. Since we can generate crash images separately for both devices, we can reuse

these heuristics and algorithms. The same arguments apply here. The algorithms are described in Section 4.3.2.

3.7 Tester

The tester combines PM and NVMe crash images into hybrid crash images based on the index created by the crash image generator. It mounts each hybrid crash image in a virtualized environment and runs a semantic state extraction on it. We extract the complete directory structure which is recognized on the crash image, including file names, file contents, and `stat` metadata: file type, inode number, file mode, number of hard links, user and group IDs, file size, number of blocks occupied, and timestamps. Semantic states are then deduplicated, and for each distinct semantic state Permanent records the set of crash images which resolved to this state. No fundamental changes had to be made to the tester compared to the Vinter version, except adding the ability to resolve which PM and NVMe images may be combined.

Chapter 4

Implementation

We now present an implementation for Permanent based on the design in the previous chapter.

Building on Revin and Vinter-Rust, our implementation of Permanent also uses Rust as its primary programming language.

4.1 Tracer

The tracer's task is to record operations on PM and on the NVMe device. Like Vinter and Revin, we use QEMU to emulate guest execution.

We need a way to observe both PM instructions and NVMe commands inside the VM. Vinter uses PANDA [2] to record NVMe reads, writes, memory fences, and flushes [13]. As a checkpointing mechanism, Vinter uses a special x86 instruction [13]. On the other hand, Revin uses QEMU's trace events that trigger when NVMe commands are executed.

For Permanent, these two approaches are unsuitable. PANDA cannot trace NVMe events, as it uses an older QEMU version without NVMe support [16]. Bringing NVMe support to PANDA would require either backporting the QEMU NVMe emulator to an earlier QEMU version, or modifying PANDA to use a newer QEMU version [16]. We deem these approaches inadequate. Furthermore, relying solely on QEMU trace events like Revin does would require substantial changes in QEMU's source code to support PM tracing, as we would have to modify the memory access code.

Instead of these two approaches, we use QEMU's TCG plugin capabilities (see Section 2.3.1). We use the plugin callback functionality to record PM operations, and combine this with NVMe operations. We use a modified version of QEMU 8.0.4. The QEMU patches are available online with the source code of Permanent: [3]

Like the other parts of Permanent, our TCG plugin is implemented in Rust, compiling to a native shared library.

4.1.1 TCG Plugin Implementation

The TCG plugin forms the heart of our tracer. It analyses guest execution flow, extracting relevant data for PM instructions and NVMe commands. We spawn a separate writer thread to receive incoming events and output them to a trace file. Like Vinter, we compress our trace with the snappy compression algorithm [5].

PM Instruction Tracing

The plugin inspects translated guest instructions and categorizes them into PM flushes, PM fences, potential memory accesses, and checkpoint instructions. For each instruction it registers an execution callback or a memory access callback, based on what information is required for this specific instruction.

TCG plugins pose a severe limitation on the kind of data that can be inspected at runtime: specifically, there exists no functionality to inspect guest memory contents and guest registers. Some instructions, namely global memory flushes and memory fences, do not require any further data. However, we need to access guest memory and registers to extract data for PM reads, writes, and fences of single cache lines.

We worked around the memory limitation by implementing guest virtual memory read and write functionality via a new plugin function. This function uses existing debugging functionality inside of QEMU. This allows us to extract relevant memory contents for PM writes. We extended this plugin function to handle physical memory reads and writes as well, because we need physical memory write access for the post-success and post-failure tracing to load crash images.

Working around the register limitation is more difficult: due to the nature of TCG translation, register values are only available at the end of basic blocks (see Section 2.3.1), which is insufficient for our instruction-granularity tracing. Register values would have to be written back after every guest instruction by inserting TCG instructions.

We bypass this problem by using memory callbacks. When tracing requires us to read a register value, we do a one byte memory read instead, interpreting the register value as a memory address. This way, a memory callback is triggered with the required register value as the virtual address, which is available to us in the callback. We do this by inserting corresponding TCG instructions into the intermediate code. This approach is limited, as it only works for register values which correspond to a valid virtual guest address without risking an address

translation error. For our application this is sufficient, because we only use it for memory flush instructions `clflush`, `clflushopt`, and `clwb`.

Checkpoint Tracing

For each checkpoint we want to transfer an unsigned value as the checkpoint ID. We do this by writing the checkpoint value into guest memory at an arbitrary location; we used a global variable. We then trigger a memory access by loading the value, which is registered for our memory callback. To distinguish this from a normal memory access, we use a signal instruction that does not otherwise appear in the guest, borrowing from the idea of Vinter's original checkpointing implementation [13]. We chose a simple four-byte move instruction with a special immediate value. Together, this signal instruction and the memory access form the checkpoint. The checkpoint code is written in C with inline assembly and compiled to a static executable. This executable is added to the initramfs for all tests.

NVMe Instruction Tracing

We cannot use guest instruction hooking for NVMe events because NVMe events are not comprised of a single machine instruction; rather, we need to extract relevant information about NVMe reads, writes and flushes at different points in the QEMU source code. To do this we modify Revin to suit our new approach: at each point in the QEMU source code where Revin generated a trace event, we call into our TCG plugin shared library instead, transferring all relevant information of the trace event.

A single NVMe read or NVMe write triggers several trace events, each carrying a subset of the relevant information. These events need to be combined into a single trace entry. Revin handles this by recording all trace events in their incomplete form, and consolidating them after the tracing is complete [16]. In Permanent, we want to do this parallel to the tracer, so the writer thread can output fully formed trace entries. Therefore, we insert incomplete trace events into a queue managed by the writer thread. This queue also contains all PM events to preserve the order of commands. Incomplete NVMe trace events are joined and, when an entry is complete, the writer thread writes it to the trace file.

TCG Plugin Configuration

There are several configuration options for our TCG plugin.

- PM start address and PM length: specify the PM address range. Any reads and writes inside this range will produce a trace entry in our plugin, while reads and writes outside this range will be ignored.

- PM base image: path to an image file which is loaded into PM before the shell command is executed.

- Trace options: specify which events shall be traced. Events not specified here will not produce trace entries. Possible options are: PM read, write, fence, flush; NVMe read, write, flush; and checkpoint.

- Trace file: file path where the trace is written to.

Note that we do not have an option to specify an NVMe base image. This is due to the fact that when creating an emulated NVMe device with QEMU, we can specify the image directly.

These options are required to implement the three different use-cases for our tracer.

1. Analysis tracing: we enable all trace options relevant for the current file system type, and checkpoints. We do not need to specify a PM base image, unless our test case is based on an already existing image. This is usually not the case.

2. Post-success tracing: we enable read trace options for the devices we are interested in, as well as checkpoint option to observe whether the tracing was successful or not. The PM base image must be specified as the current image with every write persisted.

3. Post-failure tracing: we disable all trace options, as we can extract all relevant information from the VM output (see Section 4.2.1). Most parts of the TCG plugin, like the instruction callbacks and the writer thread, can be deactivated to significantly increase performance (see Section 5.3). However, due to time constraints, this optimization was not implemented completely. We also cannot disable the plugin completely, as it is used for loading PM content. The PM base image path points to the PM part of the current hybrid crash image.

4.2 Communication with the Virtual Machine

We create a virtual machine for tracing by starting a QEMU subprocess. QEMU and plugin configuration options are passed as command line arguments. Communication with the VM is accomplished by setting up input and output pipes with QEMU's serial option, in the same way that Revin does [16].

4.2.1 Shell Command

Permanent waits until the kernel has booted and the subprocess is ready to accept commands by reading the output pipe and waiting for the last message before the shell prompt. Then, Permanent sends a single shell command to the input pipe of the VM which is used for initialization, test execution, and signaling success or failure. This shell command is constructed at follows:

- First, we execute a checkpoint instruction with the special value 255. When this value is caught in the TCG plugin, it initializes the PM area from the PM base image file. The checkpoint with value 255 is also written to the trace, so the crash image generator knows at which point the PM area has been initialized.

- Next, we execute the actual user-supplied shell command from the VM and test configuration files. This might consist of two partial commands in the case of analysis and post-failure tracing.

- As a last step, if the shell command succeeded we print a success message to the standard output, which is rerouted to our output pipe. Otherwise, we print a failure message.

4.2.2 Collecting Results

The message at the end serves two purposes. First, the parent process can read the VM output until either the success or the failure message is encountered. This way, we can be sure that we collected all VM output generated from our command. The entire VM output is written to a log file for the later stages of Permanent's pipeline and for manual inspection. Second, we receive information about whether the executed command succeeded or not. In the case of analysis tracing, we panic if the command was not successful. For post-failure tracing, after the file system dump we execute some additional user-supplied operations to verify that the file system is working correctly. Only when these operations are all successful is the the success message printed, so when the tester does not find it in the log it can map the crash image to the failed semantic state $\perp$.

After receiving the message, Permanent shuts down the QEMU subprocess gracefully by sending a signal. QEMU calls a TCG plugin exit function we previously registered, flushing trace entries to the trace file and shutting down the writer thread. There are now two output locations of the tracer: the trace file, and the VM log file. We also redirect QEMU standard error information to a separate log file for debugging, an approach we copied from Revin [16].

4.3 Crash Image Generator

Here we describe the implementation of the crash image generator.

As described in Section 3.6.1, we replay trace entries one by one onto the respective device models. Permanent panics if any PM or NVMe operation occurs before the PM area has been initialized. This is not strictly necessary, but serves as a sanity check for our test cases.

4.3.1 Generating Crash Images

Crash image generation starts when the first checkpoint is encountered in the trace. At each crash image generation point, for each device we generate the image where no unpersisted writes are persisted, the image where all unpersisted writes are persisted, and a set of random images with a subset of unpersisted writes persisted. These writes are applied on top of the guaranteed persisted content of the respective device.

The crash images are hashed with the blake3 hashing algorithm [6]. For each hash we store the corresponding crash image. We deduplicate crash images based on their blake3 hash. Furthermore, we do not distinguish crash images based on the device they belong to. This simplifies our image pool for deduplication, and has the advantage that when two devices have the same image, they are also deduplicated. However, this does not apply to our test cases, as the devices have a different size or differnt usage patterns.

4.3.2 Crash Image Generation Algorithms

We want to limit the amount of crash images generated. While it is simple to calculate the amount of total crash images possible at each point in the trace, it is hard to reason about the amount of distinct crash images that will be generated, because this depends on the writes present in the trace. We would like a mechanism to generate exactly N distinct random crash images at each invocation of the crash image generation algorithm. However, doing so would necessitate a redesign of the algorithms used in Vinter and Revin, and it may add considerable runtime overhead. For simplicity we adhere to the way Vinter and Revin limit the amount of generated images: introducing fixed parameters which act as an upper bound.

Even with these parameters we can make almost no assumptions about the amount of distinct images, because many of the generated images will be duplicates. For this reason, we implemented a size limit for the deduplicated image pool as an additional safety mechanism. When the image pool's size grows bigger than this limit, the crash image generator panics. This acts as a last resort, in

case there are more distinct crash images than anticipated. Right now the limit is statically compiled into the new Vinter executable, but could be turned into a command line parameter in a future version. The same is true for the upper bound parameters of the random image generation algorithms.

In the following, we present the crash image generation algorithms implemented in Permanent. As stated in Section 3.6.2, there is no need for Permanent to implement new hybrid algorithms.

4.3.3 PM Exhaustive Algorithm

This algorithm is used by Vinter when the amount of possible crash images is sufficiently small [8].

All writes to the same cache line are ordered in Vinter's PM model. To generate all possible crash images we look at all subsets of dirty cache lines. For each subset we look at all possible combinations of partial in-order flushes of the selected cache lines. We apply each of these combinations to the guaranteed persisted content of the PM device do generate crash images.

4.3.4 PM Random Algorithm

This is the main algorithm used by Vinter. We describe this algorithm for the simple case where there are many dirty cache lines, and many writes in each cache line. The detailed algorithm can be seen in the Vinter source code [8]. Random generation follows the same steps as the exhaustive algorithm. There are two constants S and P limiting the amount of crash images generated. First we choose S random subsets of dirty cache lines. In Vinter, this selection can also be limited by the heuristic. Next, we work on each subset individually. If for the selected lines there are less than P possible combinations of partially flushing all lines, we take all of them. Otherwise, for each line we consider two flush indices: one full flush of the entire cache line, and one random partial flush. We take every combination of these partial flushes to form crash images. We fixed a bug in Vinter which always generated all partial flush indices, even when there were more than P combinations.

4.3.5 NVMe Exhaustive Algorithm

We implement an exhaustive algorithm to generate all possible NVMe crash images when the amount of possible crash images is sufficiently small. It is adapted from Revin's random algorithm [16]. We iterate through every permutation of unpersisted writes. For each permutation, we apply the writes one after the other, and after every single write use the current state as a crash image. This approach

fully simulates command reordering and partial flushing. However, it is unsuited for real test cases, as the number of crash images without deduplication is $n! \cdot n$ for n unpersisted writes.

4.3.6 NVMe Random Algorithm

This algorithm is reused from Revin. It receives a parameter M. We create M random permutations of the n unpersisted NVMe writes, and for each permutation we create a random index from 1 up to n inclusive. We replay all writes coming before this index in the permutation, and ignore the rest to simulate partial flushing of the write cache. This algorithm creates exactly M crash images, and 1 to M distinct crash images after deduplication.

4.3.7 Heuristic

Due to time constraints we did not implement Vinter's heuristic for Permanent, although we see no difficulty in doing so. Having no heuristic does not change the correctness of our approach, but it potentially lowers the quality of crash images we generate.

4.3.8 Output

The crash image generator outputs three index files: One where trace IDs are mapped to the set of PM crash images generated at that point, and one for NVMe crash images. The third index file maps trace IDs to checkpoint values, so we can quickly see if a trace ID occurred before or after a given checkpoint.

4.4 Tester

The tester iterates through all crash images and runs the semantic state extraction on them. This is done by spawning a new Permanent tracer process. From the VM log we extract the semantic state of the file system by executing Vinter's `fs-dump` program, which writes state information to the standard output of the VM. We surround this special start and end messages to isolate it from other logging information.

For each newly encountered semantic state including $\perp$, the tester saves the full semantic state as a text file, so the user has a complete overview of the semantic states generated. These can then be manually inspected.

During execution we build a mapping of semantic state hashes to the set of hybrid crash image hashes this state originated from. After all crash images have

been examined, we output the mapping between crash image and semantic state to a JSON file.

In case we do not have a hybrid file system, i.e, we only use PM or NVMe, we can simply iterate through all crash images in the image pool, running the state extraction on each of them. On a hybrid file system, we check the index files produced by the crash image generator to find all valid crash images.

To reduce space consumption, the tester deletes all tracing-related information after post-failure tracing for each crash image. This includes the trace and the base image for this specific tracer invocation. Only the logs are preserved in case they are required for debugging purposes.

4.5 Reporting results

As a last step, we wrote a Python script to report the results of the crash image generator and tester. We do not count this as part of the Permanent pipeline, because it only exists for the convenience of the user.

The script outputs the number of PM, NVMe, and hybrid images generated. It resolves possible origin points for hybrid crash images by intersecting the set of trace IDs where the PM part was generated with the set of trace IDs where the NVMe part was generated. Combining this with checkpoint data and semantic state data, it counts how many distinct semantic states were found between each pair of adjacent checkpoints, and how many semantic states were found exactly at the checkpoints' trace IDs. The first part of this information is used for atomicity checking: if between two checkpoints there were at most two states, we report this checkpoint range as atomic. The second part is used for single final state checking: A checkpoint range is reported as single final state only if the tester found exactly one semantic state at the trace ID marking the end of this range.

Chapter 5

Evaluation

We applied our implementation of Permanent to real file systems. In this chapter we report our findings.

5.1 Testing of Non-Hybrid File Systems

We tested the NVMe-exclusive part of Permanent with an ext4 file system running a simplified version of Revin's `symlink` test case, with metadata journaling enabled. For this test, Permanent produced a total of 71 crash images, and the three expected semantic states: one where the directory is empty, one where only the regular file exists, and one where both the regular file and the symbolic link exist. This indicates that the NVMe-only part of Permanent works as expected. However, it also produced crash images which could not be mounted. Even after running fsck, our fs-dump utility panics when trying to access file metadata on these crash images. These states should not exist, as metadata journaling was enabled on the file system. We could not attribute these invalid crash images to a cause yet, but they are possible crash images based on our tracer output.

Further compatibility tests to Vinter and Revin have not been carried out due to time constraints.

5.2 Testing of ZIL-PMEM

We apply some simple test cases to a ZFS file system with ZIL-PMEM, where the ZFS pool consists of an NVMe main device and a separate ZFS intent log device backed by PM. The ZFS pool is mounted with the `sync=always` option to force writes to the devices. Our test cases use Linux kernel version 5.11. Both the NVMe and the PM device are 128 MiB in size. The pool is set up to use DAX

for the log device. We will first give a detailed look into one test case, and then show abbreviated results for all Vinter test cases.

5.2.1 chmod Test

This test case is comprised of the following shell commands:

```
checkpoint 0
echo test > /mnt/myfile
checkpoint 1
sleep 2
chmod 666 /mnt/myfile
sync
checkpoint 2
```

We traced the execution of this test case and generated up to 25 random PM images and 25 random NVMe images at each crash image generation point. Permanent generated a total of 13 PM crash images and 49 NVMe crash images, forming 61 hybrid crash images.

The tester extracts three distinct semantic states: One where `/mnt/myfile` does not exist at all, one where it has the default mode `rw-r--r--` and the expected content, and one where it has the new mode `rw-rw-rw-` and the expected content. We did not find an image where the file exists without content, which may be a valid file system state. We discuss this further in Section 5.2.2. We also did not find a hybrid crash image that led to an error when importing and mounting the file system it contained.

Atomicity and Single Final State

We find that the chmod logical operation between checkpoint 1 and 2 is atomic. To do this we examine all possible crash images between checkpoint 1 and 2. Our tester output shows that all of these crash images represent one of two semantic states: the one where the file exists with the old file mode, and the one where it exists with the new mode. The logical operation between checkpoint 0 and 1 is also atomic based on our crash images, although we did not require it to be atomic, as explained above.

We also observe that both operations are single final state. In this test case this is trivial, as there is only one hybrid crash image at checkpoint 1, and only one hybrid crash image at checkpoint 2. By implication, at each checkpoint there can only exist a single semantic state. Manually inspecting the trace shows that this is not an oversight of our random algorithm: at each checkpoint there are no unpersisted stores for either device.

Further Discoveries

Examining the trace, we find that in this test case, file contents are written directly to the NVMe device. Conversely, metadata changes like the changed file mode are written to the ZFS intent log on PM. We can derive this from the output of our tester: There is only one crash image representing the third semantic state. We find another crash image representing the older semantic state where only the PM content differs, while the NVMe content remained unchanged. This means the information leading the file system to find the new file mode is stored on PM.

Furthermore, we observe that ZIL-PMEM is well-behaved in this test case in regards to the two different devices. Writes to the PM device are non-temporal exclusively, avoiding the possibility of missing a cache flush. Additionally, ZFS with ZIL-PMEM does not modify both devices at the same time. After every set of NVMe writes there is an NVMe flush before writes to the PM device begin. Trivially, there is also a PM fence before an NVMe operation begins. We see this in the output of our crash image generator: at each crash image generation point there is exactly one PM image and one or more NVMe images, or vice versa. This has the advantage of performance benefits for the tester of Permanent: the amount of hybrid crash images grows linearly with the amount of single-device crash images, instead of quadratically.

When writing the pool device, ZFS tends to use big NVMe writes. The biggest one in this test case was comprised of 256 contiguous atomic blocks, for a total of 128 KiB. This shows the importance of NVMe write splitting as implemented by Permanent. Big writes like this cannot be simulated as being atomic without making the NVMe simulation considerably more restrictive and less accurate to real hardware.

5.2.2 Further Tests

We adapted all Vinter tests to fit Permanent. These tests execute basic file operations in the virtualized environment. Table 5.1 shows the test results. All test cases generate at most 25 random PM images at each crash image generation point, and at most 25 random NVMe images. We did not find any bugs in ZFS in our evaluation.

Differences in `ctime` and `mtime`

In the test cases `link-hard`, `rename-dir`, and `touch-long-name` we find more semantic states than expected. Many crash images resolve to a state that is exclusive to them. Examining these states, we notice they only differ in `ctime` and `mtime` for the files and directories. These times were all valid Unix

test case	#PM	#NVMe	#hybrid	#states	#states (no time)
append	17	46	62	3	
atime	1	51	51	2	
chmod	17	52	68	3	
chown	13	49	61	3	
ctime-mtime	15	48	62	3	
hello-world	1	50	50	2	
link-hard	30	46	75	20	4
link-sym	21	51	71	3	
mkdir-rmdir	18	49	66	3	
rename	11	50	60	3	
rename-dir	67	48	114	51	5
rename-long-name	30	47	76	3	
touch	20	49	68	3	
touch-long-name	48	48	95	27	3
unlink	12	50	61	3	
update-middle	12	51	62	3	

Table 5.1: Test results for running Vinter tests on ZIL-PMEM with Permanent. The number of PM images, NVMe images, and hybrid images in the table are generated distinct images, not the total number of generated images. The last column shows the number of distinct semantic states when differences in timestamps are ignored.

timestamps with a discrepancy of less than an hour between them and the time of tracer execution. The crash images producing these states share the same content except for variations in a PM range of 256 bytes.

We tested the same crash image two times, leading to two distinct semantic states. This confirmed that the changed `ctime` and `mtime` do not originate in the crash image; they are instead inserted when the ZFS pool is imported and mounted. We assume that when an incomplete log entry is detected, ZFS updates the file with the current timestamp.

From the found semantic states we removed all timestamps to see which crash images then map to the same new semantic states. The number of distinct states without timing information is also shown in Table 5.1. We find that the amount of new semantic states is consistent with what we expect from the respective test case. In conclusion, this is not a bug in ZFS, but rather a feature which we need to inspect manually. A future version of Permanent could include file system dumping without timestamps for this special case. However, in the general case, we do want complete file system information including timestamps, because inconsistent timestamps could indicate a bug in other file systems. For now, we included an option in our result reporting script which shows all distinct states after removing timestamps, and the mapping from checkpoint ranges to the possible new semantic states the crash images may represent in this range.

Missing Semantic States

In the `hello-world` test case we find two semantic states: one where the file does not exist, and one where it exists with the specific content. However, we have two basic file system operations: creating the file, and writing to the file. In total there could be three semantic states, the third one including the file without content. We find this behaviour when we modify the `hello-world` test case to execute a sync directly after mounting. In this case, the file contents are now stored on PM instead of NVMe and the Permanent tester also finds the third semantic state.

This behaviour could be caused by Permanent not generating enough NVMe crash images to find all valid semantic states, but it could also be the case that ZFS combines the file creation and file write to NVMe into a single atomic transaction.

5.3 Performance

Using TCG plugins with callbacks imposes a significant slowdown on the tracer. Because we do not decide at the time of callback installation which guest instructions lead to a memory access, and we do not know which memory operations ac-

cess the PM range, we have to install a memory callback for every x86 instruction. Even if most of them will return immediately, we have inserted TCG instructions and slowed down the translation and execution. A boot of our ZIL-PMEM kernel takes 170 seconds, and 18 seconds without installing callbacks, exhibiting a slowdown factor of almost 10.

We modified the post-failure tracing to avoid installing memory callbacks, as post-failure is not concerned about reads or writes. This was especially important because the slowest part of Permanent's pipeline is the tester, as we have many tracer invocations. The performance increase for the tester was the same as for the tracer itself.

The total execution time of our chmod test case was 41 minutes and 57 seconds on a dual core Intel i5-2520M processor, with 36 minutes spent in the tester.

Chapter 6

Discussion

In this chapter we relate the results of our evaluation to our goals and original design, and show where future work is needed or possible.

6.1 Goals

We evaluate our implementation based on the goals introduced in Section 3.1.

Permanent is able to handle NVMe-only file systems, which we verified with our test in Section 5.1. Further tests are necessary, however. We also didn't check if the PM-only part of Permanent produces results that are consistent with Vinter's results.

Because Permanent adheres to Vinter's and Revin's design, it can test atomicity and single final state for hybrid file systems as well.

Regarding completeness of our approach, for the individual devices we inherit any simplifications made in Vinter and Revin, for example Revin's assumption that flushes only happen after previous writes have been executed [16], or Vinter's approximation of a non-volatile write by flushing the respective cache line. Relevant discussion about these algorithms can be found in [13] and [16], respectively. A detailed view of Vinter's crash image generation algorithm can be found in the source code [7, 8].

In our ZIL-PMEM tests, we find all expected file system states in the test cases we applied; we also do not find states ZIL-PMEM cannot recover from. This indicates that we extended the device model in a way that is representative of the expected behaviour of PM and NVMe hardware. It is, however, not a conclusive proof. We did not attempt to recreate a known bug in an existing file system, like Vinter did [13].

Our approach suffers when it comes to performance. Vinter reports testing time for several full tests to be approximately 24 minutes [13]; our implementation

is significantly slower even when accounting for differences in hardware. We attribute this to the fact that we use TCG plugins, slowing down translation in a way that PANDA handles better. We also did not implement loading VMs from a previous snapshot, which would reduce testing time. In our tester, we boot a new VM for every crash image.

6.2 Further Work

Here we list possible future work on Permanent.

6.2.1 Omitted Features

Some features of Vinter were not implemented in Permanent due to time constraints.

- Our tracer currently cannot support metadata tracing. Vinter traced the current stack frames for trace entries, so the user could understand where in the code a specific trace entry originated. This is not possible with TCG plugins, so far, as there is no functionality to read registers. This would also mean inserting more TCG instructions to dump registers.

- We did not implement Vinter's heuristic for Permanent. It is up to future work to implement it and measure the effectiveness on it for hybrid file systems. One could also limit the application to one of the two devices, perhaps the one whose behaviour is closer to journaling, as Vinter's heuristic is most effective there [13].

- We did not include extended crash image metadata in our crash image generator. In Vinter, this metadata was used to better comprehend the origin of the crash image, including information about which cache lines were flushed [13]. In Permanent, we only know the trace ID where the crash image originated.

- As stated above, Permanent does not have functionality to start VMs from a snapshot.

6.2.2 Performance

We see great potential in reducing Permanent's execution time. These approaches could also be implemented in Vinter or Revin.

- Parallelism inside the tester. As the tester forms the performance bottleneck for Permanent, an obvious step is to parallelize it. This is trivial, as separate invocations of post-failure tracing are independent of each other. Doing so would yield a nearly perfect speedup.

- Parallelism across tracer and crash image generator. We could start the crash image generator as soon as the tracer starts producing trace entries. However, the speedup is minimal compared to a parallel tester because even the sequential execution time of the tracer and crash image generator is a fraction of the tester's execution time. Also, we want to verify that the analysis stage succeeded before generating crash images.

- Parallelism across crash image generator and tester. We could start with post-failure tracing as soon as the first crash images are generated in the crash image generator. Then these two pipeline stages could run in parallel. The same argument applies here: as the execution time of the crash image generator is small compared to the tester, we would not get a big speedup. For bigger test cases, this would have even less impact.

6.2.3 Further Improvements

Another improvement would be decreasing the amount of space the crash images occupy. We think this is possible in the following way: instead of saving all crash images, we only save the guaranteed persisted content at each crash image generation point. Additionally, we save the following crash metadata for each crash image: the trace ID where it originated, as well as the set of writes applied on top of the guaranteed persisted content. This set could simply contain indices into the trace, making the representation very compact. The tester can then reconstruct crash images from the guaranteed persisted content and the additional stores. This means only the crash images currently tested need to be physically present on the user's system; after testing they can be removed immediately. This strategy would increase the runtime of the tester, but greatly decrease the amount of space needed to store crash images.

We could also make the amount of crash images generated at each point proportional to the amount of unpersisted writes at this point. Because the set of possible crash images grows with the amount of unpersisted writes, the probability of finding distinct crash images by randomly generating them grows as well. This could even be made device-local, so each device generates a different number of random crash images.

6.2.4 Three-Tiered File System Testing

There are file systems using more than two devices; Strata [14] uses PM, SSDs, and HDDs to build a system with high throughput and high storage capacity. Permanent could be extended test file systems like this by having two or more separate NVMe devices. One would have to revisit the device model to specify how the different NVMe devices relate to each other.

Chapter 7

Conclusion

In this work we showed that it is feasible to combine Vinter and Revin into Permanent, a cross-media file system tester. Built upon previous device models, we designed a hybrid device model describing the interaction between PM and NVMe devices based on real hardware behaviour. Permanent uses this model to simulate a cross-media system. In our prototype implementation of Permanent, the added theoretical complexity of having two devices did not lead to a substantial increase in the complexity of the implementation. We reused or adapted core parts of the Vinter and Revin pipeline, as well as the algorithms they use for crash image generation.

Executing test cases borrowed from Vinter with ZIL-PMEM showed that Permanent is capable of handling existing file systems, making Permanent suitable for testing future cross-media systems involving PM and NVMe. Unfortunately, as the tracer had to be redesigned to support both devices at once, Permanent's flexibility comes at the cost of reduced performance.

To conclude our work, we described where further work is likely to lead to improvements for Permanent. This includes performance improvements to bring Permanent closer to Vinter. Also, research is needed to determine how much Vinter's heuristic could increase the quality of crash images generated by Permanent. Lastly, further work is also necessary to increase Permanent's ease of use for finding the causes of crash consistency violations by reintroducing metadata generation in the tracer and the crash image generator.

Bibliography

[1] Introduction to programming with Intel® Optane™ DC persistent memory. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-programming-with-persistent-memory-from-intel.html. Accessed: 2023-09-22.

[2] Panda user manual. https://github.com/panda-re/panda/blob/dev/panda/docs/manual.md. Accessed: 2023-09-22.

[3] Permanent: Persistent memory and NVMe non-deterministic tester. https://github.com/luwae/permanent. Accessed: 2023-09-22.

[4] QEMU's documentation. https://www.qemu.org/docs/master/index.html. Accessed: 2023-09-22.

[5] Snappy compression implemented in Rust (including the snappy frame format). https://github.com/BurntSushi/rust-snappy. Accessed: 2023-09-22.

[6] the official Rust and C implementations of the blake3 cryptographic hash function. https://github.com/BLAKE3-team/BLAKE3. Accessed: 2023-09-22.

[7] Vinter crash image generator. https://github.com/KIT-OSGroup/vinter/blob/atc22-artifact/vinter_rust/vinter_trace2img/src/pmem.rs. Accessed: 2023-09-22.

[8] Vinter persistent memory. https://github.com/KIT-OSGroup/vinter/blob/atc22-artifact/vinter_rust/vinter_trace2img/src/lib.rs. Accessed: 2023-09-22.

[9] Matt Ahrens, Jeff Bonwick, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. San Francisco, CA, March 2003. USENIX Association.

[10] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[11] Ilia Bozhinov. Reducing synchronous write latency with a PMEM write cache in the device mapper layer, April 2022.

[12] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, 2023.

[13] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. Vinter: Automatic Non-Volatile memory crash consistency testing for full systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 933–950, Carlsbad, CA, July 2022. USENIX Association. `https://www.usenix.org/conference/atc22/presentation/werling`.

[14] 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, pages 460–477, New York, NY, USA, 2017. Association for Computing Machinery. `https://doi.org/10.1145/3132747.3132770`.

[15] NVMe Express, Inc. *NVM Express® NVM Command Set Specification*, 2022.

[16] Daniel Ritz. Crash consistency testing for block based file systems on NVMe drives, September 2022.

[17] Christian Schwarz. Low-latency synchronous IO for OpenZFS using persistent memory, June 2021.