Karlsruhe Institute of Technology

# Analyzing Persistent Memory Crash Consistency of WineFS with Vinter

Bachelor's Thesis
submitted by

cand. inform. Paul Wedeck

to the KIT Department of Informatics

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

13. Juni 2023 – 13. Oktober 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, October 13, 2023

# Abstract

Persistent memory (PM) is a recent storage technology. Contrary to classical storage devices, PM has a different persistency model that adds new challenges in ensuring that each persisted state is valid. This is especially relevant for file systems because they are supposed to remain consistent regardless of random crashes.

WineFS is a novel PM file system. It attempts to retain a high performance for aged file systems. Based on the assumption that hugepages improve the file access performance, it features a novel memory allocator that attempts to minimize fragmentation and preferably serve allocation requests with hugepages. We used the testing framework Vinter to analyze WineFS for crash consistency. Vinter traces a test sequence and simulates crashes and their results at potentially interesting positions. Crash images are generated based on the persistency model of the platform, to ensure that each crash state could actually happen.

To aid our analysis, we modified Vinter in several aspects. We added a mechanism to load precreated PM images and generate code coverage for the tested file system. Additionally, we achieved significant performance improvements using output compression and multi-threading. During our analysis, we discovered several minor and two potentially critical bugs and we propose fixes to all newly discovered bugs. We confirmed all previously reported crash consistency bugs in WineFS and validated that all proposed bug fixes resolve their respective bug in our test cases.

# Contents

# Chapter 1

# Introduction

Persistent memory (PM) [5] [34, pp. 11] is a recent storage technology that provides fast, bytewise access. PM is accessed using regular load and store instructions. These unique properties necessitate a new persistency model. Block devices can usually write a single block of at least 512 bytes atomically. On PM, only eight bytes of memory can be stored atomically. Stores to PM may be reordered both by the compiler and the processor. Further, a memory barrier must be used in ensuring that a store has been persisted. Because a system may encounter a crash at any time, the exact persistency semantics of modifying a storage device are very important to ensure that the stored data is always consistent after a crash. Due to its alternative persistency model, achieving crash consistency is more challenging on PM than on regular block devices. File system are supposed to remain consistent over long times. Therefore, it is critical that they are not corrupted by system crashes [15, ch. 35].

WineFS [18] is novel PM file system based on PMFS [16]. Its main design goal is to improve the long term performance of file mappings. This is achieved by an alternative memory allocator and a per-CPU journal. If a file is mapped into userspace, the mappings may be done using hugepages. The novel memory allocator attempts to serve as many allocation requests as possible with aligned 2MB extents that may than be used as hugepages. Access to a file that is mapped using hugepages causes less pagefaults and is therefore faster. WineFS provides two consistency modes. The so-called strict mode guarantees that all file system operations are atomic and synchronous while the so-called relaxed mode does not guarantee that write operations are atomic.

The goal of this Bachelor's thesis is to analyze WineFS for crash consistency bugs using Vinter. Additionally, we evaluate if Vinter is suited for testing new file systems. Vinter [23] is a testing framework designed to test crash consistency of PM applications. It has been successfully used to find multiple bugs in PM file systems like NOVA and PMFS. Vinter traces the execution of a series of PM operations and generates simulated crash states at potentially interesting points. Then it attempts to recover from this crash and extracts the application state at this point in time. This allows users to determine if an operation generated unexpected intermediate states which indicate a crash consistency issue.

To aid our analysis, we modified Vinter in several aspects (see Chapter 4). We added code to extract additional file system state and improve the performance of the extraction process. Further, we improved the overall performance of Vinter using multithreading. To evaluate our code coverage, we added a mechanism to generate coverage reports of the tested file system during Vinter tests. During our analysis, we discovered two potentially critical and various minor bugs. Except for two regular bugs (one critical and one minor bug), all bugs can solely occur due to crashes and can be reliably reproduced using Vinter. Additionally, we reproduced various previously published [30] crash consistency bugs in WineFS. In Chapter 5, we describe all reproduced and newly discovered bugs and evaluate our Vinter modifications.

# Chapter 2

# Background

## 2.1 Crash Consistency

Data on storage devices is expected to survive over a long time despite power losses or system crashes [15, ch. 42]. Such events may happen at any time. It is therefore necessary that the persisted state is at all times consistent. A state transition is called atomic [15, ch. 26] if no intermediate state between the original and the target state is visible. The amount of memory that can be stored by atomic operations is often too limited for practical use. For example, Intel Optane PM only guarantees that writes of up to eight bytes are atomic [5]. Under such guarantees many basic file system operations cannot be executed atomically [23].

Transactions [15, ch. 26] are an alternative method to execute multiple operations as a single atomic operation. However, hardware support for transactions is not generally available [6, 36]. Software implementations, on the other hand, are possible with one technique being journaling.

**Journaling** Journaling [15, ch. 42)] prevents inconsistent intermediate states by means of an additional auxiliary data structure called the journal. Before any modification of the actual data is executed, the operation is first written to the journal. Journaling is employed by several file systems including WineFS [18].

Only when all modifications contained in a transaction were successfully written to the journal, the actual modification is executed. This can be ensured by a write barrier that prevents all following write operations from executing before all previous operations are completed.

If the system crashes before the journal entry has been successfully written, the crash recovery code may ignore the entry. In this case, the recovered state is the previous state. If the system crashes at a later point, the crash recovery code will attempt to execute all modifications included in the journal entries. Therefore, the recovered state fully includes the operations described in the journal. If the initial state was consistent and the operation in the journal maintains consistency, the data structure will always recover to a consistent state.

Alternatively, a file system might choose to write the previous data to the journal and then use this knowledge to undo the operations in the journal. This variant of journaling is used by WineFS. [18]

The crash recovery code can determine if a certain journal entry is complete in different ways. Methods include a special end marker that is written after the actual entry. However, this requires a additional memory barrier. More sophisticated file systems may include a checksum over the rest of the entry in each journal entry. This way, an incomplete journal entry is automatically inconsistent and can be safely skipped without jeopardizing data consistency.

## 2.2 Intel Optane Persistent Memory

Intel Optane persistent memory [7] is a specific implementation of persistent memory by Intel. The memory modules are in DDR4 DIMM form factor. They can be used both in so called memory mode and app direct mode. In memory mode, the module functions as additional memory [34, p. 298] without requiring explicit application support while in app direct mode, the module functions as additional storage (PM). On Intel processors, Optane is accessed like regular memory. Some file systems offer a feature called DAX which provides direct access to files stored in PM via memory mapped files [5].

**ADR** Systems supporting Intel Optane PM require Asynchronous DRAM Refresh (ADR) [6]. ADR ensures that all pending writes queued at the memory controller are successfully executed in case of a power failure. Under ADR, it is necessary to first flush a modified cache line and then insert a memory barrier to ensure that the modified data is persistent. Extended ADR (eADR) additionally guarantees that CPU caches are successfully flushed in case of a power failure. eADR, therefore, eliminates the need for flush operations to ensure crash consistency and simplifies PM access.

**Transactional Memory** Transactional memory [11, p. 1445] is a CPU extension allowing applications to execute multiple memory operations transactionally. The transaction maintains a read-set and a write-set at cache line granularity. The read-set contains all cache lines that were accessed by load instructions while the write-set contains all cache lines that were accessed by store instructions. If the memory contained by the read-set is modified or the memory contained by the write-set is accessed in any way from the outside, the processor aborts the transaction. If the transactions succeeds, the changes in the write-set are visible atomically.

As transactional memory is implemented at the cache layer, ADR guarantees are not sufficient to maintain crash consistency. On the other hand, eADR guarantees that such transactions are committed atomically [5, 36].

## 2.3 WineFS

WineFS [18] is novel PM file system based on PMFS. Its main design goal is to improve the performance of aged file systems.

On PM, memory mapped files are significantly faster than traditional read-/write system calls. Hugepages generate less page faults and TLB misses than 4KB pages. Therefore, an accessing program may experience increased performance if a memory mapped file is mapped via hugepages. Allocated memory can only be mapped as a hugepage if it consists of extents of appropriate size and alignment.

Depending on the order and location of memory allocations and deallocations, the free PM regions may become fragmented into small extents. This makes it increasingly difficult to allocate contiguous and properly aligned chunks of memory [15, ch.16], resulting in reduced performance.

**Allocator** WineFS introduces a novel memory allocator that attempts to increase the number of hugepages that can be allocated in aged file systems by proactively minimizing fragmentation. This allocator keeps a pool of aligned and unaligned memory. The aligned pool contains memory suitable for hugepages while the other contains the remaining memory.

Memory allocations are broken down into a multiple allocations of 2MB or smaller. 2MB allocations are served via the aligned pool, always resulting in hugepage mappings. The allocator first attempts to serve smaller allocations with the unaligned pool. Only if this is not possible it will use memory from the aligned pool.

On deallocation, the file system attempts to merge adjacent unaligned memory regions and if possible convert it into new aligned regions.

WineFS protects its metadata with an undo-journal. Contrary to log-structured file systems, a journaling file system reuses previous memory. Therefore, a journal requires less memory allocations and causes less fragmentation.

WineFS provides two consistency modes. The strict mode guarantees that all file system operations are atomic and synchronous while the relaxed mode does not guarantee atomic write operations.

# Chapter 3

# Related Work

## 3.1 Vinter

Vinter [23] is a tool to automatically find inconsistent crash states in PM applications. It has been successfully used to discover bugs in several file systems including NOVA and PMFS.

Vinter uses dynamic binary translation to record accesses to PM and certain events relevant for crash consistency like fences and cache line flushes. This requires no modification to the source code of the file system. It has a testing pipeline of multiple programs starting with the tracer. First, the tracer records a so called pre-failure trace that contains all writes to PM and all relevant serialization and persistence events. Additionally, the traced program may also issue checkpoint hypercalls that separate different semantic operations.

This trace is then analyzed by the crash image generator to generate different PM states that could have been restored after a system crash. Possible crash states depend on the assumed memory persistence model. When assuming the x86 model, stores are only guaranteed to be persisted after certain ordering points. Vinter generates new crash states at these ordering points by only partially applying in-flight changes. Care must be taken that the applied subset is actually allowed under the memory model. Specifically, on x86 operations on the same cache line are ordered with respect to each other [32]. Otherwise, the generator may generate crash states that could not exist on a real system, leading to false positives.

Depending on the trace, the amount of possible crash states may be too large to analyze. Therefore, the crash image generator uses a heuristic to only generate crash states that are relevant for crash consistency.

These crash states are then passed to the tester. The tester then runs on the crash recovery code and a program to extract the semantic state for each trace. If any one these trace is not recoverable, the sequence of operations was not crash consistent.

The tester can also analyze multiple traces for stronger crash consistency guarantees. For example, it can test if all crash states at one checkpoint result in the same state or if all crash states between two checkpoints result in only the original or in a single final state. It is, however, not guaranteed that multiple intermediate or final states indicate a violation of the crash consistency guarantees of the operation.

**Test Specification** Each Vinter test is based on a test file and a virtual machine (VM) file. A test file specifies which file system operations should be performed. A VM file specifies how a virtual machine with a specific file system may be started. This includes specific arguments how the VM is configured, where the kernel image is located, and where the PM region is inside the address space of the VM. Additionally, the VM file provides the VM command that is used during crash state generation. The initial trace command and crash state extraction command consist of a prefix specified in the VM file and a suffix specified in the test file. The prefix is commonly a file system specific command to mount the file system, while the suffix is commonly test specific. For the trace command, the suffix is usually a list of operations that may lead to inconsistent crash states. The suffix of the state extraction command is a test specific command to output the visible state of the file system.

## 3.2 Chipmunk

Chipmunk [30] is an alternative approach to detect crash consistency bugs in PM file systems. It has already been successfully used to find bugs in multiple file systems including WineFS.

Chipmunk does not trace PM accesses directly but traces calls to certain kernel functions that file systems use to write to PM. Functions calls are traced via the Kprobes and Uprobes debugging mechanisms which requires no source code modifications. However, this approach requires that these function calls may not be inlined by the compiler.

Chipmunk generates test cases via Automatic Crash Explorer (ACE) and Syzkaller. ACE exhaustively generates sequences of file system operations with a certain length based on a pre-defined structure. Syzkaller starts with a set of initial test cases and then generates new ones based on code coverage gathered by previous runs.

Similarly to Vinter, Chipmunk generates possible crash states based on the persistence model. If the number of possible crash states at any ordering point is small by some measure, Chipmunk generates all possible crash states. Otherwise, only a subset of all possible crash states is generated.

Chipmunk tests the consistency of each crash state by comparing the state of the file system with known valid post-crash states. Then it attempts to create and subsequently delete a file in all directories, validating that the file system is in a usable state. If any of these tests fails, Chipmunk outputs a bug report for this crash state.

### 3.2.1 WineFS

The authors of Chipmunk have found at least three confirmed crash consistency bugs in WineFS. They also found another possible bug. Further analysis will be part of this thesis. All confirmed bugs occurred in the write system call.

One bug was caused by a missing store fence after updating file metadata. This means that the new size and last-modified time of the file may not have been serialized when the system call returns.

Another bug was caused by a missing cache-line flush after writing data at an unaligned starting address. If a write to PM is not aligned to eight bytes or has a smaller size, WineFS will use a store instruction that goes through the cache. Specifically, it incorrectly determined that if the write extent does not end on an eight byte boundary, it is only necessary to flush the last cache-line if the total size of the write operation is not aligned to the length of a cache-line [20].

WineFS uses a B-tree per inode to store which blocks represent which part of the content of the inode. If this B-tree contained just one data block before a write operation and the write operation does not require additional blocks, this data block would not be marked as copy-on-write. If the system crashes while only parts of the write operation have been executed, the crash recovery program will not revert these changes. Therefore, such a write operation is not atomic, violating the atomicity guarantee of the strict mode [21], [22, inode.h, inode.c and pmfs.h]. [29]

# Chapter 4

# Approach

Our goal is to analyze the crash consistency of the file system WineFS using Vinter. Vinter is a testing framework that allows tracing file system operations and simulates crashes at certain points. Based on this information, we can decide if a crash during a certain operation may lead to an invalid file system state. Crash consistency is very important for a file system because users expect it to retain its content over long times. Losing at least parts of the content of the file system due to a simple system crash is not acceptable.

WineFS and Vinter are specialized on persistent memory (PM). PM is a recent storage technology that allows byte-granular access to its data using regular load and store instructions. Traditional storage devices are accessed based on blocks using special commands. Due to its alternative access method, PM introduces new challenges in ensures that a file system remains consistent after a crash.

## 4.1 Vinter Modifications

We decided to modify Vinter in several aspects to better suite our needs. Due to long wait times, we improved the performance of Vinter using multi threading. Further, we introduce a mechanism to starts tests using a prepared file system image instead of a clean image. We experienced issues with the system clock of the VMs created by Vinter. To mitigate these issues, we propose changes to make the generated timestamps of each file system more consistent between each VM instance and between different test runs. Vinter has to extract the file system state of each generated crash state. We improved both the performance and amount of data generated in this process. This enabled us to detect the bugs described in Section 5.3. Finally, we introduce a mechanism to generate coverage reports about the tested file system. This allows us to determine how much of WineFS we already covered with our tests.

Most of these changes were not strictly necessary to detect or verify any bug we discovered. However, they helped us analyzing WineFS. It was a common occurrence that we executed Vinter in short succession after making small changes to the test files or the source code of WineFS. Particularly the performance improvements made this process more stringent and less time consuming.

### 4.1.1 Multithreading

Testing a file system using Vinter is inherently parallel because the different tests do not interact or interdepend. Vinter instances can run fully independent of each other, as long as each instance has a distinct test file and VM file combination. However, each individual Vinter instance is single threaded and runs one VM instance at most. The VM runner creates multiple threads but we observed that it effectively only utilizes a single core. This leads to poor core utilization and unnecessary wait times when running fewer tests than CPU cores are available. During our experiments, it was a common occurrence that we worked on a single test case.

The work process was that we executed Vinter, analyzed the results, and then made minor modifications to the test parameters. We repeated this process numerous times. The most time consuming part of this process was waiting for Vinter to finish. Therefore, the performance of Vinter proved to be a bottleneck.

**Parallel Semantic State Extraction** Vinter extracts the semantic state of each generated crash image. Using deduplication, Vinter drastically reduces the amount of crash states presented to the user. However, Vinter has to run the state extraction step for each crash image. While the deduplication requires at least some synchronization, the state extraction itself can be parallelized. Further, the deduplication step is just a single hash table access and therefore a negligible overhead.

**Implementation** Vinter uses a snapshot mechanism to speedup VM load times. A snapshot is recorded once per Vinter test run and then resumed multiple times. The VM requires exclusive access to this snapshot. Therefore, we create a separate copy of this image for every spawned VM.

The state extraction is executed in a separate process. Vinter has to spawn a new process and wait for its completion. A naive implementation might spawn multiple processes and wait for their completion in some order. This leads to theoretical performance losses if the processes terminate in a different order than our arbitrary wait order. Under Linux, a process is generally capable of waiting for any of its children to complete [12, wait(2)]. However, Vinter is written in Rust and we are not aware that the process API of Rust [3] exposes this feature.

As far as we are aware, Rust only support waiting for the termination of a specific child.

Therefore, we decide to use a thread pool. We submit a new task to this pool for every external process. This task than waits for the termination of the child. Because we start the processes outside of the thread pool, we use a semaphore to limit the number of concurrently running child processes.

**Evaluation** Using parallel state extraction, we achieved a speedup of the state extraction phase of up to factor 20 for 32 cores. A few tests with particularly short extraction times only achieved a factor of 1.9 or 3.4. If we sum the duration of the average extraction time for all tests with and without parallelization, we achieved a speedup of 18. This shows a clear performance improvement for this specific stage.

Overall, we achieved much lower speedups. A run of all Vinter tests is faster by a factor of 2.6. Selected tests achieve speedups of up to factor 6. In general, tests that run longer benefit more from parallel semantic state extraction. This highlights that this modifications is qualified to reduce wait times where this issue is the most pressing.

**Parallel Test Execution** Parallel semantic state extraction has the issue that the core utilization of each Vinter instance fluctuates. Running the tests sequentially leads to poor core utilization during the significant single threaded part of each run. As each test is highly multi threaded, running the tests in parallel might lead to performance degradation because the maximum number of active threads is significantly higher than cores are available. Under this assumption, we decided to execute multiple test files in the same Vinter instance to better manage core utilization. We achieved this by creating a single worker thread pool sized by the number of processor cores or a user-defined value. As a first step we queue the first, sequential part of each test run on this thread pool. Once all tests have been queued, we wait for the sequential part to finish and run the previously parallelized part of each test run.

In our tests, this did not lead to any significant performance improvement over naive parallelization.

**Crash State Generation** For an unmodified Vinter version, the state extraction phase clearly determines the overall execution time. Using parallel semantic state extraction, the performance determining factor becomes the crash image generation phase.

Crash images are currently generated iteratively. While some level of parallelization might be possible, it would require significant modifications to the existing code. Introducing new bugs to this part of Vinter risks generating illegal crash states or hiding crash states that indicate the presence of crash consistency bugs. Due to previous performance improvements and the risk and effort involved, we did not further pursuit to parallelize state generation.

### 4.1.2 Aged Images

A major design goal of WineFS is to improve performance of aged file systems. Specifically, it attempts to maintain a pool of memory regions that may be used for huge pages. This implies that WineFS may behave differently and thereby introduce new bugs on aged file systems.

#### Implementation

We introduced a `load_pmem` parameter to test configurations and VM configurations. The parameter in the test file overwrites the parameter in the VM file if present. This allows to both base test cases on specific initial states and executing arbitrary test cases on an aged image.

Vinter uses the `vinter_trace.py` script to start a VM. It already has a functioning parameter to load initial PM states. This is used for crash state generation and semantic state extraction. We use this parameter to load an precreated PM image in the initial tracer run.

Further, we modify the crash state generator to use this image as the initial state of our PM. This is necessary because the VM only records the modifications to the PM but not the initial state.

#### Full Images

To test the behavior of the memory allocator if it has only few pages available, we created a custom file system image. We first create two dummy files, then a file that fills approximately 1/3 of the file system and then a dummy directory with a single dummy file in the directory. All dummy files had a size of a few bytes. Then we created a file that allocated all remaining memory. Finally, we deleted all dummy files and directories. This ensures that the file system has enough memory for at least two files and a directory with one file and each file may have at least a page of content.

We used this mechanism to execute all existing Vinter tests on this file system image. Thereby, we discovered the unlink bug. We described this bug in Section 5.6. This bug is in its nature not dependent on the general file system state. However, with a clean file system image, our test case only generates a single illegal intermediate state. Using our aged file system, it generates varying numbers of illegal intermediate states.

### 4.1.3 FS-Dump

Vinter comes with a tool to extract the state of a file-system called fs-dump. During each state extraction process, fs-dump is called to extract the state of each crash state. We observed that this process experiences significant performance degradation for large files. Files filled with null bytes cannot be prevented when testing allocate and truncate operations. Further, fs-dump prints the escape sequence `\u0000` for every null byte. Therefore, the output size of such files is highly inflated. This worsens the original performance issue if such files are present.

Most Vinter tests do not create particularly large files. Therefore, this is not problematic for most regular tests. We still experienced performance degradation in some cases. Further, we observed that the performance issues are mostly based on the size of the output.

#### Compression

We decided to compress the output of fs-dump to represent the contents of the file more compactly. We propose a simple compression scheme that replaces consecutive occurrences of a single character with the number of occurrences and the specific character. This reduces the minimal output size to $O(\log n)$ from $O(n)$, for $n$ input characters. Such a compression scheme works well if the file is mostly filled with a single character. It is easier for the human eye to see small differences between such files, compared to files that are filled with random values. Further, such files are very easy to create. The behavior of a file system is mostly independent of the actual content of a file or a write operation. Therefore, it is only a small limitation for Vinter tests to preferably create such files.

This compression scheme has the downside that it inflates the output size if the contents do not properly fit into the compression scheme. To reduce the impact of this, we do not use this compression scheme if a character is different from its preceding and succeeding character. Such characters are combined into a string literal.

Further, we compare the size of the file and the size of the compressed contents and only output the compressed string if it is smaller. If the file size is smaller or equal in size, we output the raw file contents. Note that both string may contain unprintable characters which will be represented by an escape sequences. This may inflate the actual output size. Therefore, this heuristic might underestimate the size of both variants. In our tests, files generated during Vinter tests experience a significant reduction in output size.

For small sizes, the uncompressed content string makes up only small parts of the overall output strings. We measured that the performance overhead is negligible. Additionally, the raw output is generally more readable for such files. Therefore, we decided to not compress small files. As this is a trade off, there is no objective optimal value. We arbitrarily decided on 128 bytes as the separating file size.

### Performance Evaluation

We evaluated the performance of fs-dump by allocation a file filled with null bytes and subsequently dumping the content of the file using fs-dump. This approach enables us to create files larger than the physical size of the file system. However, it has the limitation that fs-dump outputs a null byte as 6 bytes (`\u0000`). Additionally, fs-dump outputs a static amount of metadata. For example, dumping an empty file generates 377 bytes of output. Dumping a file of $10^6$ null bytes generates $6 \cdot 10^6 + 386$ bytes of output. Other file sizes follow this pattern. Therefore, the output size can be roughly estimated by multiplying the file size with 6.

For a file filled with 500 null bytes (output size is 3381 bytes), the VM execution time using the baseline version is 7.38 seconds. The highest file size where we successfully generated results was 5623 bytes (output size 34121 bytes). This took approximately 115 minutes. The next highest test would have been with a file size of 7499 bytes but this failed to do a builtin 9999 second timeout of Vinter.

A VM execution without any tasks requires approximately 2.19 seconds. For $10^6$ bytes the VM execution time is only 2.59 seconds. Such a file is reasonably small and the slowdown is acceptable. This is significantly faster than what we achieved using the baseline version.

For larger files, the slowdown is more significant. Using our compression scheme, we tested files with sizes up to $10^9$ bytes. For a file size of $10^9$ bytes, the VM execution time was 39.6 seconds. This highlights that there is still optimization potential. However, for crash consistency testing, files of a few kilobytes to a few megabytes are most common. For such files, our compression scheme provides a significant improvement in comparison to the baseline version.

### UTF-8

An unmodified fs-dump version interprets the file content as a UTF-8 string. This means that printable characters get a reasonable representation and unprintable characters are printed using special escape sequences. However, some files do not contain valid UTF-8 data. In this case, fs-dump crashes.

This is unacceptable because some bugs (see Section 5.1) cause files to be filled with unexpected values which are not necessarily valid UTF-8. Therefore, we propose to read the file as bytes, convert each byte to a character and collect all these chars into a string. A byte may have values between 0 and 255. During the conversion to a character, this value is interpreted as a unicode codepoint. All of these codepoints are valid, therefore this cannot fail. Collecting them to a full string cannot fail either because each combination of these characters is legal. Note that this will misinterpret most texts that are not ASCII. However, such texts are rare in crash consistency testing because they offer no benefit over ASCII text.

#### Extended Attributes

WineFS supports extended attributes [12, xattr(7)]. These attributes are key value pairs that contain metadata about a file. We added code to fs-dump to list and extract all extended attributes of a file. This enabled us to discover the bugs described in Section 5.3.

### 4.1.4 Time

Vinter utilizes a mechanism to speed up VM startup by creating a snapshot once and then continuing it for every VM invocation. We noticed that the initial VM had a correct system time. However, the VM is not aware that it was temporarily halted. Therefore, its system time starts at the last moment of the previous VM instance. To verify the truncate bug (see Section 5.5) we discovered, it was necessary that the VM system time is consistent between VMs. This means that a timestamp generated at a later point in time is always younger than one generated before it.

We achieved this using the hwclock [17] tool from util-linux. QEMU provides a real time clock (RTC) by default [13]. Hwclock can force the kernel to synchronize its internal clock with the RTC. By executing such a command every time the VM is resumed, we can make sure that system time in each VM progresses and is consistent regarding other VMs.

Further, we configured QEMU to start the VM time at a predefined point in time. Therefore, timestamps do not depend on the time the test was started. However, the run time of each test depends on external factors like other concurrently running tests. Further, Vinter is not fully deterministic. Specifically, the number of generated crash states varies. Therefore, the number of VM invocations and further their timestamps will vary.

### 4.1.5 Coverage

We created a mechanism to record the code coverage of each test case. For this, we used the builtin gcov [4] coverage support of the Linux kernel.

The mechanism works as follows:
Relevant parts of the linux kernel (in our case the WineFS file system) are compiled with coverage generation enabled. After running each test, a debugfs is mounted at `/sys/kernel/debug`. Linux places generated coverage files there. These files are collected into a single archive. A prepared file of sufficient size is passed to the VM as a virtual drive. The VM copies the contents of the archive to the virtual drive. Vinter extracts this archive on the host system.

We collect the coverage for all VM invocations. These are merged using `gcov-tool` to a single set of coverage files for each test case. The coverage report mechanism was implemented on top of the Parallel Test Execution modification. Therefore, multiple Vinter tests may be execution inside a single instance. We merge all per test coverage reports into a global report for this Vinter invocation.

For reason we did not further investigate, the VM fails to resume from a snapshot if the virtual drive is present. Therefore, we disabled snapshot loading when we are recording coverage information. The snapshot mechanism can be safely disabled because it is merely a performance optimization. In our tests, the snapshot mechanism reduces the startup time of a VM from 2.186 to 0.064 seconds. While this is certainly a performance degradation, Vinter is still reasonably fast. We did these benchmark using the same system as all our other benchmarks. We described the system in Section 5.11.

**GCC** We compile Linux using the GNU Compiler Collection (GCC) [14] which includes a widely used C compiler. Gcov [14, pp. 250, ch. 10] is a tool included in GCC. A program must be compiled with special options, to generate coverage data.

Coverage information is usually stored as files in the file system. Therefore, generating coverage using gcov is transparent to most programs. Some programs like the Linux kernel do not run in an environment that provides a file system. In this case, the program has to interact with gcov to extract coverage information. The details of this process differ between GCC versions. In case of the Linux kernel, the coverage files are stored in the aforementioned debugfs.

WineFS is based on Linux 5.1 which was released in 2019 [35]. The gcov support in Linux was updated to GCC 10.1 in 2020 [1]. This makes it necessary to compile Linux with a GCC version older than 10.1.

We use a Fedora 38 system which only provides recent GCC versions. Therefore, we diverted to compiling an old GCC compiler ourselves.

### 4.1.6 Crash Image Exploration Limit

We based our work on a version of Vinter that fails to properly limit the amount of generated crash images. For every fence, Vinter chooses multiple random subsets of cachelines at which to generate crash images. For each subset, it creates multiple crash images by partially applying the writes to these cache lines.

If the number of possible combinations for a random subset is reasonably small, it generates all possible crash images. Otherwise, Vinter generates only two crash images at this location. One where the state is random and one where all modifications have been applied.

However, Vinter failed to properly test how many combinations are possible. Therefore, Vinter always tried to generate all possible crash images. This bug was initially discovered and fixed during a parallel running Bachelor's thesis.

We assumed that all existing limits in Vinter worked as intended but were insufficient at limiting the number of crash states for our use case. Therefore, we introduced a new limit. We decided to limit the number of crash images that may be generated for each subset to 100. Using this fix, we no longer experienced unreasonable amounts of crash images.

## 4.2 WineFS

We included WineFS in Vinter (see Section 3.1). This requires a kernel image and a VM configuration file. To compare different modified version of WineFS, we created multiple kernel images and VM files.

Vinter only provides a minimal initramfs [24] based on Busybox. The initramfs is an archive that provides the first user mode programs. Most VM configurations share a single initramfs. If a file system requires special files, it should have a separate initramfs. Due to the structure of Vinter, it is advisable to have only few initramfs images. Specifically, it should not be specific to one kernel image.

Therefore, WineFS should not be built as a separate module file. We adapted the NOVA kernel configuration file for WineFS. The only important change is that we disabled NOVA and instead included WineFS as a builtin module. We modified the source code of WineFS slightly to set the default log level of WineFS to verbose. This additional information has been proven useful to investigate several bugs not related to crash consistency. We described these bugs in Section 5.10.

We based our VM file for WineFS on the VM file for NOVA. We achieved this by referencing our WineFS build artifacts instead of the NOVA build and changing the file system in all commands to WineFS. Depending on the specific test run, we enabled the strict mode using the strict option.

While using Vinter, we noticed that some part of the testing pipeline hangs if a file contains null bytes. This is a common occurrence when using the truncate operation to increase the size of a file (see Section 5.5). Therefore, we sanitized each command that outputs file contents using `cat -v` [8, ch. 3.1]

Vinter uses virtual PM devices. The NOVA configuration used one with a size of five megabytes. However, WineFS failed to initialize a file system on this PM device. While investigating this issue, we added additional debug messages to `pmfs_init_blockmap` to determine why the function failed. After further investigation, we discovered that WineFS fails to initialize on PM devices which are smaller than 9 MB. After increasing the size to 16 MB, WineFS could be successfully initialized and mounted. We did not further modify the VM file.

We did not change the runtime behavior except for additional debug information. Therefore, none of our source code modifications could have affected the existence of any bug we detected or reproduced.

# Chapter 5

# Results

Our goal was to analyze the crash consistency of the novel persistent memory file system WineFS. A file system is expected to safely store its data over long periods. It is expected to retains its content even if it crashes during an operation.

Persistent memory is a recent storage technology that provides byte-wise access using regular memory instructions. Traditional storage devices are block devices which must be accessed via special commands. Therefore, PM file systems encounter new challenges in ensuring crash consistency.

We used the testing framework Vinter for analyzing the crash consistency of WineFS. For some tests, modifications to Vinter were required. Additionally, we significantly improved the performance of Vinter. While this was not required, it greatly eased our task.

We found multiple new bugs in WineFS and reported them on the WineFS issue tracker at https://github.com/utsaslab/WineFS/issues. Additionally, we successfully reproduced all previously reported WineFS bugs [30] using Vinter. In this chapter we will describe all bugs, how we reproduced them and propose bug fixes where appropriate. Further, we evaluate our attempted performance improvements for Vinter and we describe our results using the Aged Images and Coverage mechanisms. Our Vinter modifications and benchmarks can be found at https://github.com/paulwedeck/vinter.

## 5.1 cmpxchg16b

We discovered two instances where cmpxchg16b with the lock prefix is used to modify 16 bytes values in WineFS. WineFS inherits this behaviour from PMFS. Both use this to update inode fields without additional crash consistency measures. Specifically, to set the file length together with the file timestamps and the inode tree root block together with the tree height.

This is based on the assumption that the full 16 B write is persisted atomically [16]. While this might be true for some CPUs, this is not architecturally guaranteed. The movdir64b instruction is the only architecturally guaranteed way of persisting more than 8 B atomically [33].

Vinter handles a `lock cmpxchg16b` instruction conservatively and interprets it as two eight byte writes. Therefore, crash images with only the first half of the operation are generated. Crash images with only the second 8 B write are not generated because the 16 B must be in the same cacheline and x86 intra-cache line ordering guarantees that the second write must be persisted after the first one Section 3.1. If only the first half is executed, invalid intermediate states are visible and the file system operation is not atomic which violates the atomicity guarantees of WineFS.

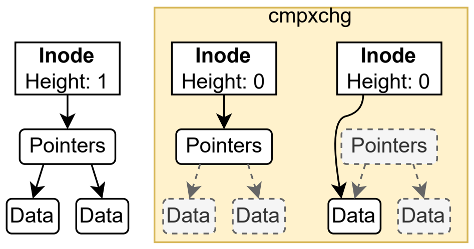

Figure 5.1: From left to right: Intermediate states that occur during a truncate operation. In this example, the truncate necessitates decreasing the tree height. If the operation fails, it is retried during recovery. The file system cannot distinguish between state two and three. If state two occurs, it assumes that the new root block has already been applied. During read operations, the file system will only traverse to the first layer, reading pointers from the inner block instead of the intended data block.

Specifically, if only the new root block is persisted but not the new tree height, the file contents are lost (see Figure 5.1). This is obviously unacceptable.

This behavior was previously reported [25] using Chipmunk but the bug was not confirmed. We investigated this issue and found its root cause based on the provided reproduction steps.

When Vinter updates the file length together with the file timestamps, the new timestamps might be lost if a crash occurs during an append operation. Tools like GNU make [9] only work correctly if the last modification time is correctly updated on every file modification. Due to clock drift and time synchronization, the file timestamps are commonly wrong or inconsistent and therefore untrustworthy.

This indicates that a fix for this bug is not strictly necessary. However, it is at least advisable that both steps are reversed, so that an updated timestamp without data modifications is observable but not vice versa. Regardless, we propose a bug fix in Subsection 5.1.3.

### 5.1.1 movdir64b

Future processors [31] will feature the movdir64b instruction which can persist 64 bytes atomically. The only precondition is that the address is aligned to 64 bytes.

Using this instruction, a possible bug fix for both issues would be to copy the first 64 bytes of the inode, modify the respective fields and persist these changes with movdir64b. This is correct for both operations because the inode is larger than 64 bytes and is aligned to 128 bytes [22, inode.h].

Because movdir64b is not commonly available, this is not a general solution.

### 5.1.2 Tree Bug Fix

The erroneous operation is only used for decreasing the height of the inode tree. The increase operation is not affected. The decrease operation is only executed in `__pmfs_truncate_blocks`.

This operation can only be safely executed if both the height and root block is updated atomically. However, the root block field is eight bytes long. It is unclear if the full 64 bit width is actually required but any changes to the data layout would break compatibility to existing file system instances and drivers. Otherwise, either a software transaction or operations that can atomically update more than eight bytes are necessary.

Therefore, we propose to wrap all inode modifications in `__pmfs_truncate_blocks` in the path where the height must be decreased in a software transaction. This transaction should only contain the inode and should be committed just after all modifications have been executed. Using this modification, Vinter no longer detects any invalid crash states.

**Alternative Approach** Crash recovery of a truncate operations works by just retrying the operation. We tested the alternative approach where we just reset the height if the root was not modified. A truncate operation should always modify both fields but not just one. Therefore, this should only do anything if we experience a crash where the cmpxchg was not fully executed. By reverting the first half, we simulate that the crash occurred just before the cmpxchg and not in the middle.

We implemented this by adding the old height and root to the truncate list. This approach bears similarities to the fix to the other truncate bug we discovered (see 5.5).

This approach has the benefit that it does not require a software transaction. However, it makes deep assumptions about how a truncate operation is executed. Future changes to the truncate code must aware of that.

### 5.1.3 Date Bug Fix

The erroneous operation only occurs if a write operation appends a file without requiring additional blocks. We propose to disable the fast write path for such writes at least in strict mode. For relaxed mode, we propose to execute the timestamp update before the file length update. This should have a lower performance impact than disabling the path completely which requires a potentially expensive software transaction. This behavior is not ideal because the only part of the write that has actually been executed is the timestamp update. However, it is not too undesirable and correct.

## 5.2 Relaxed Mode

We discovered a bug that occurs when updating the contents of a file in relaxed mode. This bug only occurs if the write goes through the fast write path which can only be the case if just a single PM page is modified.

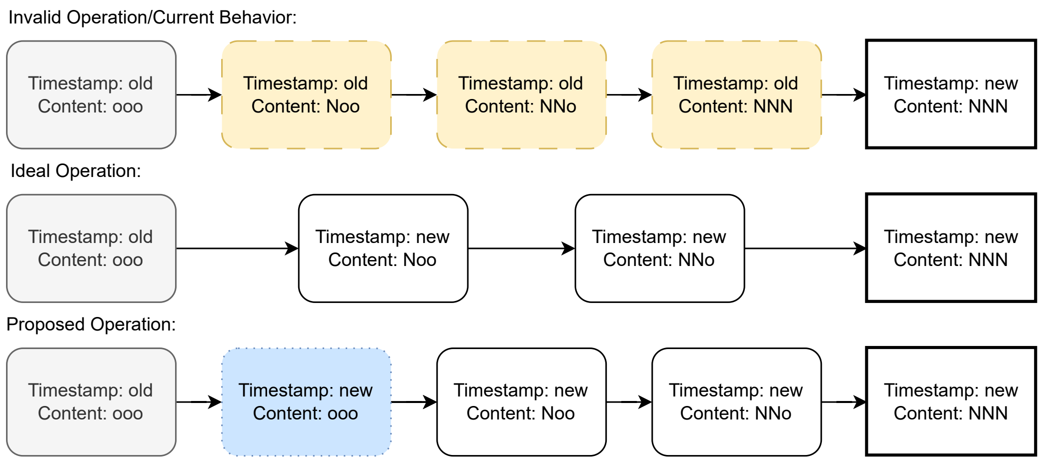
Invalid Operation/Current Behavior:
Timestamp: old Content: ooo
Timestamp: old Content: Noo
Timestamp: old Content: NNo
Timestamp: old Content: NNN
Timestamp: new Content: NNN
Ideal Operation:
Timestamp: old Content: ooo
Timestamp: new Content: Noo
Timestamp: new Content: NNo
Timestamp: new Content: NNN
Proposed Operation:
Timestamp: old Content: ooo
Timestamp: new Content: ooo
Timestamp: new Content: Noo
Timestamp: new Content: NNo
Timestamp: new Content: NNN

Figure 5.2: Visible crash states for writing three bytes to a file for an unmodified WineFS, an ideal write implementation and our proposed bug fix. The three yellow/dashed states are invalid, the blue/dotted state is valid but only occurs with our porposed bug fix.

In the fast write path, the new timestamp is written after the file contents have been modified. Therefore, crash states can be observed where the write has been partially or fully completed without updating the timestamps (see row one in Figure 5.2). We previously explained in Section 5.1 that missing timestamp updates should be considered a bug.

### 5.2.1 Bug Fix

To fix this bug, we propose to modify `pmfs_file_write_fast` by moving the code to update the vfs and WineFS inodes before the code to actually write the data. Note that this function originally consisted of two paths. One where only the timestamp is updated and one where the timestamp and file length is updated. The file length update has a similar bug which we described in Section 5.1. We proposed to move the timestamp update, if no file length update is necessary, before the code to update the inode content.

It must be noted that the inode update and subsequent write must be separated by a `pmfs_flush_buffer(pi, 1, true)` call. This flushes the inode and guarantees the persistence of all changes using a memory barrier. The result of this change is that no crash states can be observed where the file has been modified without also updating the timestamps.

**Timestamp Update** On the other hand, crash states are possible where the timestamps has been modified without any changes to the file (see row three in Figure 5.2). For this operation, we modify the mtime and ctime of the inode. The mtime is supposed to track the time of the last modification while the ctime tracks the time of the last status change. [15, ch. 39] We are not aware that WineFS gives any explicit guarantees concerning their semantics. It is explicitly stated that using relaxed mode “data operations are not atomic and may be partially completed on a crash” [18]. As updating the timestamp is part of a write operation, only updating the timestamp is a valid way of partially completing the operation.

It is also not wrong to record that this inode was modified, even though the modification only actually modified the timestamps. Without the bug fix, both timestamps are already modified if the new and old file contents are identical. Therefore, this behavior may even occur without a crash. We conclude that this behavior is correct.

**Alternative Approaches** We also considered to atomically update the timestamps together with at least some modified data. In this case, the new timestamp would atomically become visible with the first modified data (compare row two/three in Figure 5.2). However, the timestamp update itself requires writing two 4 byte values. Additionally, the file contents are stored outside of the inode. We are not aware of any hardware support to persist more than eight bytes in total to completely different physical addresses. Therefore, such a fix would require a software transaction.

While the behavior described above is not ideal, we consider a transaction to be an unreasonable overhead for such a minor issue.

## 5.3 Extended Attributes

WineFS supports only one, predefined extended attribute (xattr) [12, xattr(7)] called `file_type`. It can hold the value mmap or sys. The file type is saved in a special per inode memory region for extended attributes. Further, each inode has a boolean field called `huge_aligned_file`. The `huge_aligned_file` field is not publicly accessible but can be manipulated by modifiying `file_type`. The `huge_aligned_file` flag is also set if an allocation to the inode is larger than 2MB. If this flag is set, the file should be allocated using hugepages (2MB pages). This is a part of the novel allocator implemented in WineFS [18].

If a file that has the `huge_aligned_file` flag is copied, the new file will not automatically inherit this flag. Some programs will also copy the extended attributes of a file. A file that has the `huge_aligned_file` flag, will have the `file_type` attribute set to mmap. When the program attempts to set `file_type` to mmap on the new file, it will get the `huge_aligned_file` flag. If the xattr is set to sys, `huge_aligned_file` is set to false. Thereby, the allocator behavior for the old file is also propagated to the new file.

**listxattr** During a listxattr [12, listxattr(2)] operation on a directory, WineFS checks if each file in the directory has the `huge_aligned_file` flag set. If this condition is true, it sets the flag on the directory.

During all listxattr and getxattr operations, WineFS checks if the `huge_aligned_file` flag is set and if the `file_type` attribute does not exist. If both hold true, it creates a new `file_type` attribute with the value mmap.

If a program attempts to read the attribute before executing a listxattr operation, the attribute may not exist. Further, this flag persists even if the condition is no longer true.

It defies our expectations that a system call like listxattr modifies the external state of a file system. However, we are not aware that WineFS make any contrary guarantees.

Further, even if the file system is mounted with the read-only option [12, mount(2)], such listxattr or getxattr system call may still create the `file_type` extended attribute. If the underlying condition that lead to this decision is no longer true, the attribute persists. This is even the case if the condition has been observed while the file system is mounted as read-only.

We are of the opinion that no operation executed while the file system is read-only should influence the future external states of the file system. Therefore, we consider it to be invalid behaviour to permanently set the `file_type` attribute during listxattr and getxattr operations.

**setxattr** The setxattr [12, setxattr(2)] system call is responsible for creating and modifying extended attributes. We discovered that multiple setxattr operations are not crash consistent.

Creating an extended attribute with the value mmap is atomic. Creating an extended attribute with the value sys yields an intermediate state where the attribute cannot be read.

Modifying an existing attribute always yields two intermediate states. In one intermediate state, the attribute cannot be read. Additionally, there is an intermediate state where the file has the mmap flag but its parent directory has no extended attributes. It is unclear if this state is invalid. However, WineFS guarantees that metadata operations are always atomic. Therefore, no intermediate state may exist at all. If an extended attribute exists according to listxattr, it should always be readable. Therefore, we consider all states where the attribute is not readable to be invalid.

### 5.3.1 Setxattr Bug Fix

To fix the bugs concerning setxattr, we propose to include all data that may be modified during such an operation in a transaction. Currently, only the inode is included in the transaction and only if the attribute is first created. We propose to include the inode data unconditionally in the transaction. Further, the external memory region for extended attributes should also be included in the transaction. For a create operation, this should obviously be done after the region has been allocated. These changes require that the size of the transaction is increased by another entry.

Using this modification, Vinter no longer generated intermediate states for any tested setxattr operations. This indicates that these changes are qualified to fix this bug.

### 5.3.2 Listxattr Bug Fix

To fix the bug concerning listxattr, we propose to add a new function that checks if an inode should have an implicit `file_type` extended attribute. This function checks if the `huge_aligned_file` flag is set or the inode is a directory and all files in the directory have this flag set. If both conditions are true and the file does not yet have a `file_type` attribute, it should return true. Otherwise, it should return false.

In the WineFS handler for listxattr, we remove the code that sets the `huge_aligned_file` flag if all files in the directory have this flag set and the code that sets `file_type` to mmap if the `huge_aligned_file` flag is set. There is an abort condition that checks if the inode has no memory region for extended attributes. We extend this condition to also check if the file has no implict `file_type` attribute.

We modify the WineFS handler for getxattr by removing the code that sets the `file_type` attribute. Further, we remove the code that aborts if no memory region for extended attributes exists. We only read the `file_type` value from the extended attribute region if it exists. Otherwise, we check if the implicit `file_type` attribute exists. If it exists, we set the returned `file_type` to mmap. If neither an explicit nor an implicit `file_type` attribut exists, we return that this attribute does not exist.

This ensures that a getxattr always presents the same `file_type` value regardless if a listxattr has been executed before. Further, executing a getxattr or listxattr on a read-only file system no longer create a new `file_type` attribute on relevant files.

## 5.4 Fallocate

We discovered a bug in the fallocate [12, fallocate(2)] system call that is related to the bugs described in Section 5.3. An fallocate operation that allocates 2MB of storage to a previously empty file sets the `huge_aligned_file` flag. This flag is part of the internal state of a file but gets exposed during a listxattr system call if the `file_type` extended attribute does not yet exist. By this proxy, this flag becomes part of the external state of the file.

Fallocate requires a transaction but this transaction does not include the part of the inode where this flag is located. Further, the operation does not flush the `huge_aligned_file` flag. Therefore, this new value of the flag might not be persisted after the operation has been completed. We observed simulated crash state where the fallocate operation has completed but the `file_type` extended attribute does not exist. This violates the guarantee of WineFS that all metadata operations are synchronous.

Further, the flag might have been set even if the operation has been aborted. Spuriously setting this flag is not invalid behavior but it violates the atomicity guarantee of WineFS.

**Bug Fix** We propose to include the complete inode in the transaction used by fallocate. This ensures that the flag is flushed if the transaction is committed and any changes are reverted if the transaction fails.

## 5.5 Truncate

WineFS provides the truncate operation which reduces or increases the size of a file. When increasing the size, the new region is filled with null bytes. Using the truncate operation, we discovered intermediate states where the truncation was successful but the timestamps pointed to the recovery time and not time where the operation was executed. This occurs because inodes that are supposed to be truncated are saved in "a so-called truncate list, which is a linked list of inodes which require further processing in case of a power failure" [22, inode.c:2070]. However, these entries only contain the inode number and the new length but not the new timestamp of the inode.

This means that the time of crash recovery influences the recovered state. Such behavior does not violate any guarantees made by WineFS because the operations are under all circumstances both synchronous and atomic. However, the result of the operation differs between the normal and the recovery path. Contrary to other timestamp bugs, this is generally not an issue because the timestamp is updated under all circumstances. However, a user might rather be interested in the last time an application interacted with an inode than the last time the memory associated with the inode was modified.

### 5.5.1 Bug Fix

As a bug fix, we propose to extend the truncate list with two additional fields that each records the mtime and ctime of the truncate operation. Additionally, we add a marker field that signals if the list entry contains a mtime and ctime field. These fields should be added at the end of the `pmfs_inode_truncate_item` struct. The timestamp fields should be 32bit long because their target field has this size. For the marker field, a single byte is sufficient. The truncate items are stored behind each inode. WineFS allocates 128 bytes to each inode, regardless of the actual size of the data structures. With our modifications, additional 9 bytes to a total of 115 bytes are used. Therefore, the format is compatible with previous WineFS versions.

It must be noted that older WineFS versions will not read these regions. To prevent this bug, both the driver before and after the crash must have these modifications.

On each truncation operation, at least one timestamp is updated. We move the time that this timestamp is determined before the truncate operation. This can be achieved by moving the update code in `__pmfs_truncate_blocks` to `pmfs_notify_change` just before the `pmfs_truncate_add` call. Further, we move the ctime update in `pmfs_unlink` before the `pmfs_truncate_add` call. These operations are not relevant for crash consistency because they only update the fields in the RAM inode. The subsequent PM operation is executed in the same way but with potentially insignificantly older timestamps.

We write the new mtime and ctime to the truncate list entry just after the truncate size in `pmfs_truncate_add`. Right after this, we set the marker to a non-zero value.

Finally, we check the marker flag in `pmfs_recover_truncate_list` just above the `pmfs_setsize`. If the flag is zero, the entry contains timestamp information. In this case, we read the new mtime and ctime from the truncate list and write it to the RAM inode. Otherwise, we use the current time as mtime and ctime. This resembles the original, bugged behaviour. If we just did not modify the inode, the timestamp would not change at all, hiding that the inode was modified.

This has the potential issue that WineFS versions without this bug fix might write values other than zero to the marker. We did not experience this behavior in our tests. However, this is a general issue with modifying the data structures of WineFS, as we are not aware of any guarantees concerning the content of unused memory regions.

## 5.6 Unlink

We discovered a crash consistency bug in the unlink operation where a crashing unlink operation updates the timestamps of an inode. As this operation is supposed to be atomic under WineFS, only the initial state or a state where the operation has completed is legal.

If the unlink operation unlinks the last link of the inode, WineFS writes an entry to the truncate list. This truncate list entry truncate the inode's size to its current size. The other modifications in the unlink operation (including the decrementation of the link counter to zero) are executed inside a transaction. If the file system crashes some time after the truncate list entry has been written, the transaction may have been completed or rolled back.

During recovery, the truncate list is processed. If the unlink transaction has succeeded, the inode link counter is now zero and the inode will be deleted. Otherwise, the truncate operation is executed which is supposed to not modify the inode. However, the truncate update also updates the timestamps of the inode to the current system time. Therefore, crash states are visible were the inode exists but has been spuriously updated.

### 5.6.1 Bug Fix

We propose to only update the size of an inode in the truncate list, if the new size differs from its current size.

This can be achieved by modifying the `pmfs_recover_truncate_list` function. Currently, the size update is only executed if the `i_nlink` field of the inode is not zero. We propose to add the condition that the `i_size` field of the inode is not equal to the truncate size of the entry. Note that the truncatesize is stored in little endian and should be first converted to the system endianess if this is not little endian.

When combining this bug fix with the fix for the truncate bug (see Section 5.5), the code for the timestamp update of the inode must naturally be guarded behind the size condition as well.

## 5.7 Issue 1

The WineFS bug `Issue 1` [28] was caused by a missing fence in the fast write path. Such a write always consists of writing some data to the file and a following metadata update. The metadata update was correctly written and flushed. However, the strict mode guarantees that each operation has completed when the system call returns. This requires that a fence is executed after the last persistent memory operation, which was not the case.

By design, Vinter tests are commonly a series of shell commands containing checkpoint hypercalls. Most operations (e.g., write, truncate) require a sequence of multiple system calls, commonly starting with an open and ending with a close system call. Therefore checkpoints only occur after a sequence of system calls. However, WineFS guarantees that each individual operation is atomic and synchronous [18].

In this specific case, under some circumstances, a following close operation executes the missing fence instructions, thereby hiding the bug from a naive Vinter test. To better match the crash consistency guarantees of WineFS, a checkpoint must be inserted after every WineFS related system call. To reproduce this bug in Vinter, we decided to write a custom program that explicitly executes each necessary system call interleaved with checkpoint hypercalls.

With this approach we can reproduce the issue. Further, we can verify that the previously published bug fix [19] correctly works for our test case. Therefore, we do not propose a bug fix of our own.

While it is possible to reproduce this bug in Vinter, the general design of Vinter makes specifying such tests complex. Vinter is designed to test file systems on the level of shell programs. However, file systems like WineFS define their crash consistency semantics on the level of system calls. Vinter tests neither stronger nor weaker guarantees. If two write system calls are executed during a shell command, WineFS guarantees that each is atomic. However, Vinter can only test if the full operation including both writes is atomic. This is a stronger atomicity guarantee than WineFS provides.

In our case, the fence executed by a close system call is falsely accounted to the preceding write. Therefore, Vinter falsely assumes that the operation is synchronous. To better fit tests to the guarantees made by the file system, Vinter should be extended to allow specifying test cases on a system call basis.

## 5.8 Issue 3

The WineFS bug `Issue 3` [26] was caused by a missing cache line flush. This bug occurs when writing a multiple of eight bytes at an address that is not eight byte aligned. In this case, WineFS erroneously does not flush the last, partially written cache line. Therefore, the content of the last cache line might not be persisted.

This violates the guarantee of WineFS that all operations are synchronous in strict mode. Additionally, crash states where the data in the last cache line was not modified are possible. This is invalid in strict mode. In relaxed mode the same behavior occurs. This is legal behavior. However, even a following synchronization system call [12, sync(2), fsync(2)] is unable to persist the requested modifications. Therefore, this bug is also present in relaxed mode.

**Reproduction** Reproducing this bug using shell commands is difficult because the bug only manifests for very specific write system calls. The specific amount and order of write system calls is very important for testing the correctness and crash consistency of a file system. However, for most regular use cases, the specific write calls are an irrelevant implementation detail. Therefore, only few commands allow specifying a preferred write size and most commands do not make any guarantees regarding their write behavior at all. We discovered an edge case in the dd command of Busybox v1.35.0 where it uses the preferred block size as write size if both the source and target is a file. This enabled us to reproduce this bug.

**Bug Fix** We can verify that the previously published bug fix [20] solves this issue for our test case. Therefore, we do not propose a bug fix of our own.

## 5.9 Issue 5

The WineFS bug `Issue 5` [27] manifested in an operation where parts of a file were overwritten. The normal write path in WineFS creates a transaction where the affected old blocks are deleted and replaced with new blocks. Then, the new content is written to these blocks. If this transaction aborts, the new blocks are again deleted and the old blocks with their old content are still present. The data blocks are stored in a tree. If this tree has height zero, the single block is not replaced but modified in place. Therefore, a transaction abort cannot roll back the modification [22].

**Bug Fix** This behavior can be observed using the preexisting Vinter test `test_-update-middle` which creates a $2201 < 4096$ byte long file and overwrites six of these bytes using one write call. Specifically, we can observe a total of five intermediate states where only parts of the second write call are executed.

Additionally, we can verify that the previously published bug fix [21] no longer leads to any invalid intermediate states, indicating that it successfully fixes this bug. Therefore, we do not propose a bug fix of our own.

## 5.10 Other Bugs

We discovered two additional bugs that are not related to crash consistency. Both are caused by incorrect error propagation. Specifically, we triggered them by executing file system operations on a WineFS instance that had no remaining storage capacity.

### 5.10.1 Write

One bug manifested in the write system call. The Linux write system call may, for various reasons, write less data than requested. Therefore, it must be executed in a loop until either an error occurs or the full operation is completed. If no further bytes can be written, the write system call must either return zero or indicate an error. When a write operations return zero once, subsequent writes must either actually write some bytes or return an error [12, write(2)]. In our case write infinitely returns zero. Programs expecting conforming file systems may therefore experience infinite loops.

This bug can be reproduced by first creating a new file and allocating all available memory to this file. This can be achieved using the `fallocate` system call or shell command [12, fallocate(2)]. Subsequent append operations (e.g., via the $>>$ shell operator) fail due to lack of available memory. This bug also manifests if the allocate operation leaves an aligned hole of at least 4096 B and a subsequent file system operation attempts to write to this region, requiring a memory allocation.

The allocation failure is not correctly propagated to the write function. Therefore, the write function only reports that zero bytes have been written. This can be experienced using the BusyBox v1.35.0 version of the dd command or the command `echo abcd >> file` with `file` replaced by the prepared file.

**Bug Fix** As a bug fix to this issue we propose to modify the WineFS function `pmfs_find_and_alloc_blocks` such that in case of an error the variable `err` is returned. Additionally, the function `__pmfs_xip_file_write` should set `status` to the result of `pmfs_get_xip_mem` if it is negative. A negative result of `pmfs_get_xip_mem` implies than an error occurred which already aborts the write operation. However, if no bytes have been written the functions returns the error code stored in `status` which otherwise remains zero, falsely indicating no error.

### 5.10.2 Fallocate

The other bug manifested in the fallocate system call. To reproduce this bug, one must create a new file and allocate all available memory to it. Next, create a new, empty file. A subsequent attempt to allocate 4097 B bytes to this new file will crash the kernel.

Like PMFS, WineFS uses a B-tree to manage the data in each inode [16]. For file sizes up to 4096 B the height of this tree remains at its initial value of zero. It is necessary to increase the height for larger allocations. This can be achieved by calling the WineFS function `pmfs_increase_btree_height` with appropriate arguments.

Increasing the height requires allocating blocks for the inner nodes of the tree. This will fail because we previously drained the file system of free blocks.

While `pmfs_increase_btree_height` correctly stops on an allocation failure, it does not propagate this error to the calling function `__pmfs_alloc_-blocks`. This function subsequently attempts to populate the tree with blocks using `recursive_alloc_blocks`. The id of the root block is initialized to zero and was not modified because allocating it failed.

Therefore, `recursive_alloc_blocks` attempts to get the memory behind the block with the id zero which is interpreted as an invalid reference and is therefore mapped to the null pointer. When `recursive_alloc_blocks` subsequently attempts to traverse the tree, it attempts to read from the root block. This results in a null pointer dereference in the kernel which kills the calling process.

**Bug Fix** As a bug fix to this issue, we propose to modify `pmfs_increase_-btree_height` to propagate a negative return value of `pmfs_new_blocks` to its caller. This can be achieved by returning the variable `errval` if its value is negative. Otherwise, the function should return zero to indicate no error.

## 5.11 Performance Improvements

We modified Vinter in several ways to reduce wait times while running crash consistency tests. To evaluate the impact of these changes, we did several performance benchmarks.

A Vinter test is fully specified by a VM file and a test file. These files further reference an initramfs and a kernel image. To maintain comparability between different Vinter variants, we used the same set of files for all tests. We used an original initramfs image.

For most tests, we used the time [12, time(1)] command to record the execution time of each program. In this section, the execution time is “the elapsed real time between invocation and termination” [12, time(1)] of a program. In a few cases, we used integrated time measuring functions to determine the execution time of each phase of a program. In this case, we use the sum of the execution times of all phases as a proxy for the overall execution time.

Further, for some tests, we recorded the CPU time each program spends in user and kernel mode. The CPU time is the sum of the time the program spends on each processor core. By adding the user and kernel space CPU time, we determine how long the program spends running on the CPU in total.

All tests were executed on a Fedora 38 system with two 16 core Intel Xeon E5-2630 processors and 64 GB of RAM. We used two SATA SSDs as storage. During our tests, no other significant unrelated load was on the system.

We compare three different versions of Vinter. The first version serves as a baseline for our benchmarks. It has no modifications that are relevant for performance. We test the version `opt1` which implements parallel semantic state extraction. Further, we test the version `opt2` which implements parallel semantic state extraction and parallel test execution as described in Subsection 4.1.1.

### 5.11.1 Test Quality

We did tests to verify that the execution time of Vinter does not change significantly between test runs. For this, we recorded the execution time of each test case using the baseline version.

We measured that the execution time varies only slightly for most test cases. It is visible in Figure 5.3 that few test cases form multiple clusters with low variation.

After further investigation, we can verify that the execution time of most test cases is highly dependent on the number of generated crash states (see Figure 5.4). Because Vinter is not fully deterministic, the number of crash states can vary between test runs and therefore the execution time of each test varies as well. However, if we sum the execution time of all n-th executions of each test, the total time does not form clusters and has a reasonable variance. All effects that we will further describe have a significance beyond this margin of error.

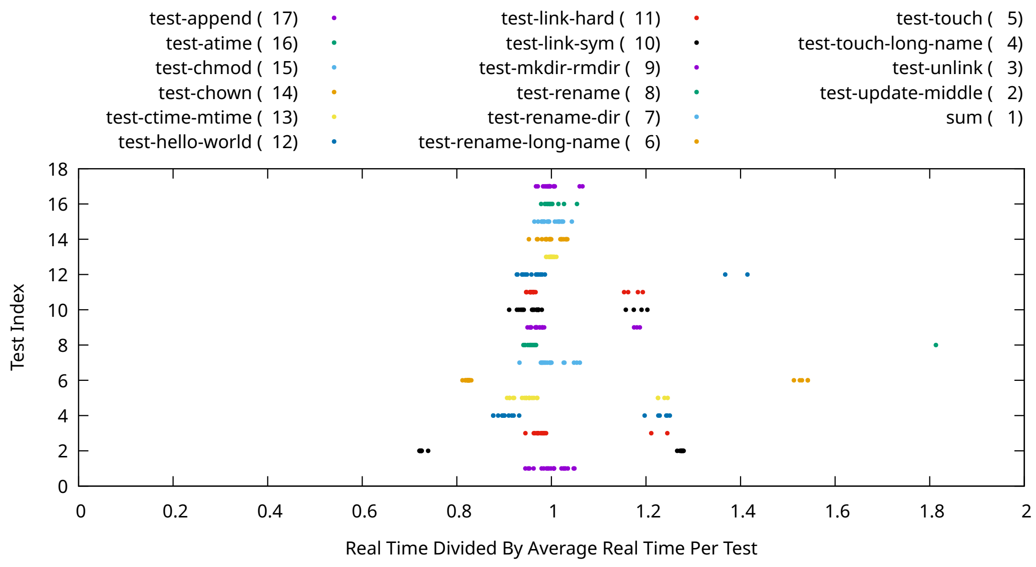

Figure 5.3: Relative execution time of twenty test runs for all Vinter tests. The execution time form one or two clusters which itself have limited variance.

## 5.11.2 Vinter CPU Usage

Using the data generated in the aforementioned tests, we compare the execution time and CPU time of each test. Note that these tests were all executed on the baseline version. We created a linear regression to predict the CPU time using the execution time. This regression is $T_{cpu} = 0.97 \cdot T_{real} + 0.79\,\text{s}$. The $R^2$ metric indicates how well the regression explains the relationship. It ranges from $0$ to $1$. We achieved a $R^2$ value of $0.992$. This indicates that our model is reasonably accurate to describe the relationship between execution time and CPU time.

The execution time of our shortest test was $11$ seconds. Therefore, the linear term dominates the model. Further, the CPU time and execution time is in general almost identical. Only small parts of Vinter utilize multiple cores at all and at each time the CPU usage is dominated by a single thread. We conclude that

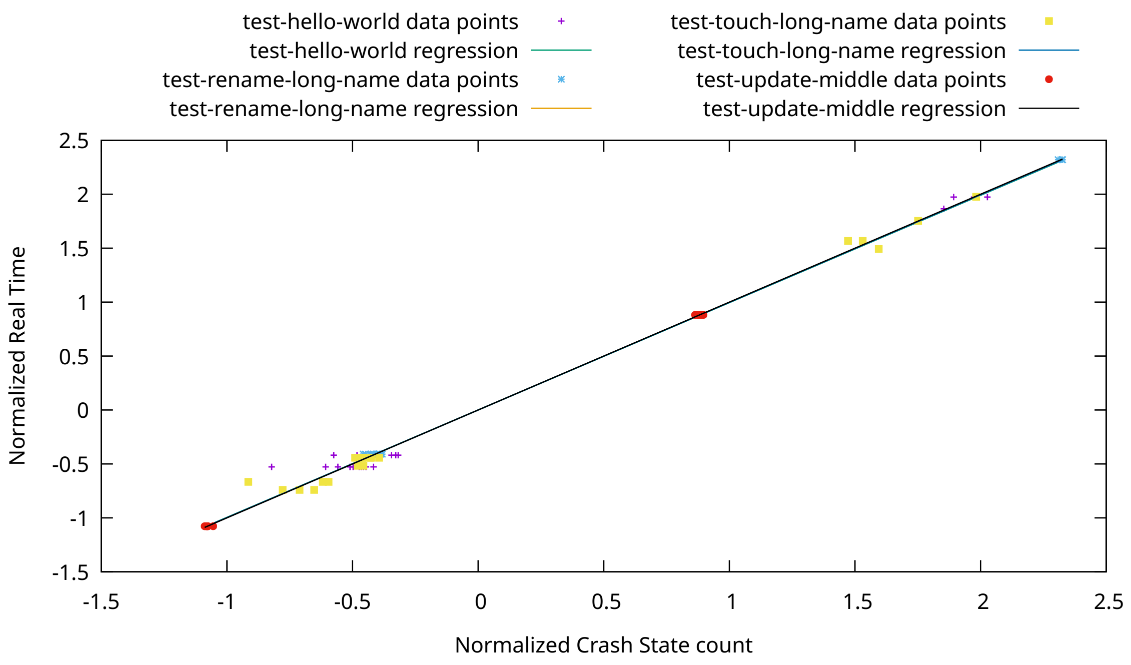

Figure 5.4: Normalized plot of number of crash states and execution times for selected tests for twenty test runs. There is a strong correlation between execution time and number of crash states. The crash states form multiple clusters. The data was normalized by subtracting the average and dividing it by the standard deviation.

each Vinter instance only effectively utilizes a single CPU core. Therefore, it is a reasonable approach to execute multiple Vinter instances in parallel to improve the CPU usage and reduce wait times. The baseline version of Vinter already features example scripts to execute Vinter instances in parallel.

### 5.11.3 Vinter Execution Phases

Each Vinter execution consists of the phases pre failure trace, crash state generation, and crash state extraction. We analyze the relative impact of each phase in the baseline version, to determine to optimization potential.

We characterize a phase as dominating if the execution time of long tests is mainly caused by this phase.

Figure 5.5 clearly shows that in our tests, the execution time of the trace phase is mostly static. It ranges from 5 to 8 seconds. No correlation between a long trace phase and an overall long test is visible. The overall execution time of each test ranges from a few seconds to multiple minutes. While this phase can certainly be optimized, the overall impact for the baseline version is negligible for most tests. Relevant speedups may be achieved for some faster tests. However, performance improvements for these tests are less relevant.

Particularly for long tests, the state extraction phase dominates overall performance. It is therefore reasonable to optimize this phase first. We accomplished this with Parallel Semantic State Extraction. This was implemented in `opt1`.

The state generation phase makes up small parts of the overall execution time for the baseline version. Note that there is a correlation between a longer generation phase and a longer overall execution time. This is to be expected because generating more crash states should take a longer time and more crash states cause a longer extraction phase. Further, we already describes that a higher number of crash states correlates with a longer overall execution time.

Figure 5.6 shows that using `opt1` the dominant phase is the state generation phase. Therefore, further performance improvements should be directed towards this phase. The state extraction phase still makes up relevant parts of the overall execution time of most longer tests. The trace phase becomes more relevant for the overall execution time. However, the optimization potential for this phase is still limited and not particularly significant for longer tests.

### 5.11.4 Naive Parallel Test Execution

To measure the impact of naive parallelization, we executed all Vinter tests in parallel. Because we already determined that the execution time of Vinter is reasonably stable, we did only five test runs. The average execution time was 615 seconds with a standard deviation of 15 seconds.

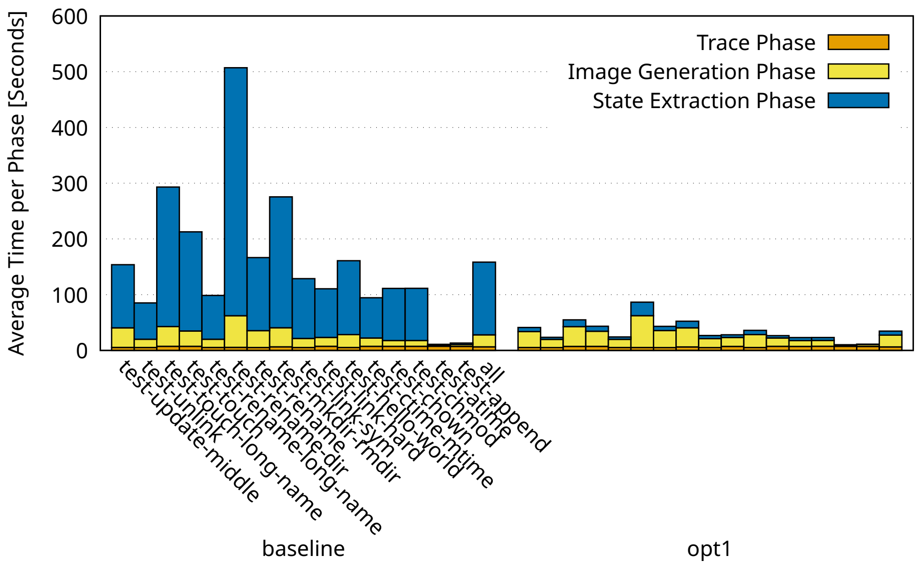

Figure 5.5: Average execution time of each phase per test. The state extraction phase clearly dominates the overall execution time. Using opt1, the state extraction phase and overall execution time is clearly reduced. (15 iterations per test and version)

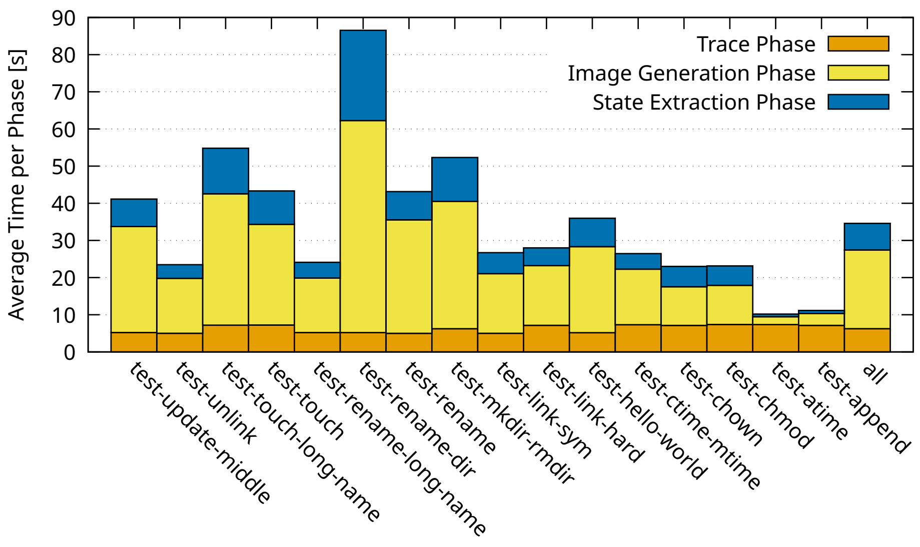

Figure 5.6: Average execution time of each phase per test. The state extraction phase is clearly faster than using the baseline version. The crash image generation phase is now clearly the dominant phase. (opt1 version, 15 iterations)

The average sum of the execution time of all Vinter tests was 2507 seconds with a standard deviation of 78 seconds. While this is a significant improvement, we would expect much higher improvements.

For a computer with 32 CPU cores, we would expect multi threading without additional optimizations to reach a maximum speedup of 32. We would expect this approach in particular to yield a maximum speedup of the number of parallel processes. Because we use 16 tests, we would expect a maximum speedup of 16.

We expect an attempt at process-level parallelization to be limited by the longest execution time of each individual process. The longest average execution time of a test was 523 seconds for the test `test_rename-dir`. Compared to the sequential run time of 2507 seconds, the maximum speedup for this set of test cases is quite limited. As we already achieved an average execution time of 615 seconds, we already exhausted most of the optimization potential.

### 5.11.5 Parallel Semantic State Extraction

To determine the performance impact of parallel semantic state extraction, we measured the execution time of each Vinter test using the version `opt1`. We executed this test five times. The average sum of the execution time of all Vinter tests was 560 seconds with a standard deviation of 29 seconds. As we previously mentioned, sequentially executing all test takes 2507 seconds on average. This is a significant improvement and faster than process-level parallelization.

Further, we executed all tests using the version `opt1` in parallel. This further reduces the execution time to 228.9 seconds with a standard deviation of 3.5 seconds during five tests. This is a speedup of more than factor 2.6 compared to process-level parallelization using the baseline version.

We also compared the individual execution time of different tests. In selected tests, we reached a speedup of 6.

The state extraction phase itself often experienced a much higher speedup (see Figure 5.5). However, relevant parts of Vinter are still single-threaded. Therefore, this optimization is not qualified to cause comparable global speedups.

### 5.11.6 Parallel Test Execution

Executing highly multi threaded programs in parallel may lead to performance degradation due to high CPU contention. Therefore, we propose to execute all tests in the same process with a unified thread pool. With this approach, we reached an average execution time of 223.16 seconds with a standard deviation of 4.1 seconds. We executed a total of five tests. This is not a significantly different result compared to executing `opt1` in parallel. We conclude that this modification does not lead to noticeable performance improvements.

### 5.11.7 FS-Dump

We evaluated the performance differences between a baseline fs-dump version and a modified version that implements our content compression scheme. These tests were executed on the same computer as our Vinter tests. We compare the execution time of the `vinter_trace.py` tool for different commands and initramfs instances. The baseline initramfs uses a baseline fs-dump while the compression initramfs contains an fs-dump that implements our content compression scheme. During our tests, we did not use any advanced Vinter features like PM tracing or snapshot loading.

We measured the execution time the VM required to execute an empty shell command. During 20 iterations, we measured an average baseline time of 2.185 seconds with a standard deviation of 0.04 seconds.

Figure 5.7 clearly shows that our content compression scheme provides a major speedup for most test cases. Overall, our fs-dump version induces no significant overhead for small file sizes and a tolerable overhead for reasonably large files. Note, that files larger than the backing storage of their file system are not common in Vinter tests.

For very large file sizes, the execution time of our fs-dump version followed a quadratic curve. We measured the maximum VM execution time for this version at $10^9$ bytes with 39.56 seconds. During our tests, the generated files were significantly smaller. If a test requires such large files, additional optimizations might be advisable.

In comparison, the baseline version of fs-dump Between 5kB and 7kB, our benchmarks started to fail due to a builtin $9999$ second timeout of Vinter.

## 5.12 Aged Images

We executed all existing Vinter tests using an image which had only a small amount of available capacity left. While creating this image, we discovered two WineFS bugs that are not related to crash consistency. We described this bugs in Section 5.10.

We first noticed the unlink bug (see Section 5.6) using the aged image mechanism. Vinter discovers more invalid intermediate states when using an aged file system than using a clean image. However, we later noticed that this bug also occurs with a clean file system. Therefore, this mechanism was not strictly necessary to discover this bug.

These out-of-memory bugs occured in common use cases and specifically on the first allocation in each operation. This indicates that this aspect of WineFS is mostly untested. Therefore, it may be promising to test more complex series of

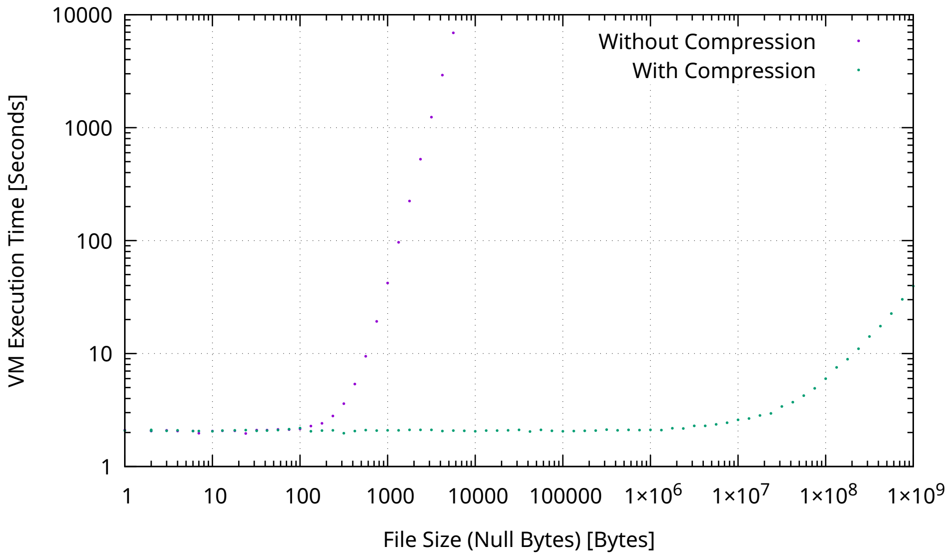

Figure 5.7: Execution time of the Vinter VM while creating and dumping files of different sizes using two fs-dump versions (with- and without compression). The version with compression is significantly faster. The data for the version without compression is limited by a builtin Vinter timeout of 9999 seconds. Note that the output string of the VM is approximately 6 times larger than the file size.

operations or allocator states to uncover additional bugs.

## 5.13 Coverage

Using our coverage mechanism described in Subsection 4.1.5, we analyzed the code coverage of our tests for WineFS. Some WineFS bugs only occur in the relaxed mode while others only occur in the strict mode. Therefore, we generated our coverage data using two Vinter runs (one for each mode) and then merged the coverage data.

We achieved a total line coverage of $71.9\%$ for the WineFS code base. This value does not include code in the Linux kernel not related to WineFS.

Such a low code coverage indicates that relevant parts of the code base are untested. Therefore, we inspected which parts of WineFS are untested and if further tests for these code regions should be done.

If the superblock of a WineFS partition has the wrong magic number or mismatches its checksum, WineFS attempts to restore it using a redundant copy. Damage to the superblock might corrupt the whole file system. Therefore, such a recovery mechanism is very useful. We originally achieved no coverage on this mechanism which is unsurprising because the superblock should not be corrupted during normal operations. Therefore, we created a new test that determines the crash consistency of this feature. To trigger the mechanism, we copied random data to the start of the file system. Using the logs generated by WineFS, we can verify that this triggered the mechanism. However, we did not discover any invalid intermediate states.

Additionally, WineFS supports multiple file system features that we did not test:

- We discovered that code regarding redo journaling is untested. WineFS inherits this code from PMFS but does not use this kind of journaling [18]. Therefore, covering this code is neither relevant nor achievable for testing WineFS.

- We did not test any code related to mmap. Using mmap, the application is responsible for most crash consistency aspects. Further, the crash consistency guarantees of WineFS for mmap and related operations are unclear.

- WineFS provides several ioctl [12, ioctl(2)] commands. These commands are not standardized and we are not aware of any further documentation. Therefore, it is unclear what their intended crash consistency guarantees are.

- WineFS is an exportable file system [2]. This is a special interface that enables referencing files independently of their current name and location. Such references are required for remote file systems like NFS. While testing this feature is certainly interesting, it is complicated to achieve using the current Vinter setup. Further, it raises the question what the crash consistency semantics in such a situation are.

- The lseek system call [12, lseek(2)] provides options to detect holes in a file which are not backed by storage and are implicitly filled with zeros. This enables programs to skip these regions while reading a file. Even if a region is reported as backed by storage, it might still be filled with zeros. While it is not mandatory for a file system to support these options, WineFS provides support for them. Vinter only considers the content of a file to be part of the file system state. There is no clear semantic under which circumstances a file system may or may not fill a hole (e.g., add zeroed backing storage to the file region).

  However, WineFS guarantees that each file system operation is atomic and synchronous in strict mode. Therefore, it might be interesting to test if all operations that fill a hole execute this atomically. Is it, however, questionable if such a bug has any further implications because filling a hole in a file only modifies the state representation while the file retains its semantic state.

- When storage is allocated to a file, WineFS preferably uses PM from the same NUMA node where the file access occurred [18]. Vinter uses a VM which emulates a single CPU core and that naturally only has a single NUMA node. Therefore, we did not achieve any coverage of code related to NUMA-awareness. However, we doubt that this would uncover additional crash consistency bugs because the NUMA code does not functionally alter any PM operation.

- We did not achieve code coverage on various mount options. Some of them are related to features that we did not test for other reasons. Further, none of the untested mount options have any relation to crash consistency. Therefore, we see no reason to create additional tests for them.

- WineFS has support for remounting the file system. We did not test this functionality. However, the only code unique to remounting writes the new mount time to the superblock. This field is not exposed and therefore not relevant for crash consistency. In general, remounting a file system should not alter its state and the code does not indicate that it does change the file system state. Therefore, it is questionable in what way this operation may be tested.

- Linux provides an interface called direct I/O [12, open(2)]. WineFS supports this interface but it uses the same functions like regular read and write operations. Therefore, it is highly unlikely that testing WineFS using direct I/O would uncover crash consistency bugs that are unique to this interface.

In conclusion, further tests are certainly possible

## 5.14 Vinter

During our analysis, we created various Vinter tests. This process is very error-prone because Vinter has severe issues with tests that spuriously run a long time or do not terminate at all. We discovered that the VM terminates by a builtin 9999 seconds (more than 2 hours) timeout if the program output is larger than a few kilobytes or contains null bytes. During our tests, a successful VM execution terminated within a minute, usually even within a few seconds.

Vinter provides no information to distinguish between a naturally long running Vinter test and one that will eventually fail due to the VM timeout. We gathered this information by manually monitoring how old each child process of Vinter was and then terminating the Vinter run if a child process got to old. However, Vinter VMs only print their output after they successfully terminated. Therefore, no terminal output is generated by VMs that fail. Why a VM execution hanged can only be judged by experience or by manually executing the VM with only parts of the workload. By escaping null bytes and compressing the content of large files, we were able to address most of these issues for our tests. However, this only mitigates the underlying issue because other programs might still output null bytes.

Vinter tests are a series of shell commands. For write operations, a user has only limited control on how many file system operations are used by each command. However, precise control of each operation is very important for testing file systems because they define their crash consistency semantics on this level. If a shell command decided to split a single large write operation into many small ones, Vinter will correctly discover one intermediate state for every write system call. This is not what was intended when creating the test and it severely increases the execution time of such a test. Further, Vinter tests should only contain as few operations as necessary because each additional operation causes additional crash images which slows down the testing process.

To verify our bug fixes, we created various kernel images. Vinter requires a new VM configuration per kernel image where only the path to the kernel image is modified. If any other aspect of the VM configuration must be modified, it is necessary to manually modify all VM configurations. While this is not ideal, during our tests, we did only few modifications to our VM configurations. Therefore, this is only a minor issue.

To understand how an invalid crash state is created, it is necessary to determine a corresponding crash image, the exact store operations that lead to the state, and the file system functions that executed these stores. This process is mostly manual. Further, Vinter only allows to dump all stores that have been executed which is usually a very large list which must be manually reduced to only the relevant section. A crash image is always generated at a specific fence. The most relevant extract of the store trace could be automatically generated by outputting all stores between the previous and current fence and highlighting the writes that are part of the crash image.

While Vinter has several aspects that require unnecessary manual engagement, the output of Vinter was always reliable. We did not experience any issue with Vinter where invalid crash states did not occur or illegal crash images were generated.

# Chapter 6

# Conclusion

It is very important that the data stored in a file system is consistent even if the system crashes during a file system operation. We analyzed a novel persistent memory (PM) file system called WineFS for crash consistency. Persistent memory is a recent storage technology that provides fast, byte-wise access.

WineFS [18] provides the crash consistency guarantee that all metadata operations are always atomic and synchronous. Synchronous means that the modifications done by all operations that have completed before the system crashes are visible after the crash. It provides the so-called strict mode where data operations are also atomic and synchronous. Further, WineFS provides the so-called relaxed mode where data operations may be partially completed and are not necessarily synchronous.

Our goal was to to verify if WineFS conforms to these guarantees and if not, to modify WineFS accordingly. We used the testing framework Vinter which traces a series of file system operations and simulates crashes at potentially interesting positions.

We discovered and fixed various crash consistency bugs in WineFS using Vinter. Additionally, we were able to reproduce all crash consistency bugs in WineFS that have been previously discovered using Chipmunk [30]. Further, we were able to confirm and fix a previously unconfirmed bug in WineFS (see Section 5.1). We published all bugs discovered by us and their fixes in the WineFS issue tracker.

We analyzed all bug fixes proposed by us or other people on the WineFS issue tracker for crash consistency. For each bug, we applied the respective bug fix to WineFS and rerun the Vinter test we created to reproduce the bug. In all cases, Vinter no longer generated any invalid crash states. By its nature, Vinter cannot prove the absence of potentially invalid crash states [18]. However, this result strongly indicates that all proposed bug fixes are qualified to fix their respective bug.

The WineFS issue tracker contains multiple bugs without any relation to crash consistency. We did not further consider these bugs nor their respective fixes because our focus is on crash consistency.

While we discovered various bugs in WineFS, only few bugs are critical. We discovered two non crash consistency related bugs (see Section 5.10). Both bugs occur because of failed error propagation if WineFS runs out of storage. One bug leads to a null pointer dereference in the kernel which terminates the calling process which is obviously critical. The other bug leads to processes infinitely retrying to write to a file. This might still severely impact programs that would usually behave well if the file system runs out of storage space.

Most of the crash consistency bugs we discovered leads to invalid, spurious, or missing timestamp updates. While this is clearly erroneous behavior, timestamps are notoriously unreliable. Therefore, these bugs are not critical.

We detected several bugs in the extended attributes support of WineFS. Under some circumstances, these attributes exist but cannot be read. This might cause issues for programs that are not well behaving or not particularly robust for unexpected file system states. Extended attributes are an optional file system feature. Further, some file systems like WineFS only allow specialized attributes with predefined values. For these reasons, a program generally should not depend on extended attributes and only few do. Therefore, the impact of these bugs is limited.

We further analyzed a previous unconfirmed bug (see Section 5.1). This bug may lead to data loss which is obviously critical. However, existing physical machines provide stronger atomicity guarantees. On these systems, this bug cannot occur.

We conclude that WineFS still has many minor bugs but only few critical ones. Further, all bugs only concerned an inode which was modified during the test. Even if any of these bugs is triggered, the damage to the file system is minimal and the overall data integrity is still ensured. This indicates that WineFS still has to mature but is already reasonably stable.

With few exceptions, an unmodified Vinter version would have been capable to detect our newly discovered bugs. However, Vinter has severe issues with tests that spuriously run a long time or do not terminate at all. Most of these issues occur because Vinter fails to properly dump the file system state if the generated file contains null bytes or the files in the tested file system are larger than a few kilobytes. Further, even if Vinter works correctly, the overall performance is poor. Therefore, Vinter tests must be specifically designed to terminate at all and to not create to many unnecessary crash states to keep the performance at a reasonable level. While we addressed these issues to some degree, these are still relevant design considerations which hinder the main goal of using Vinter.

While the test creation process has some issues, Vinter is still very capable of detecting crash consistency bugs, once a test case has been created. In Section 6.1

we describe multiple Vinter modifications that would have further improved the capabilities of Vinter. While further improvements to Vinter should be pursuit, Vinter is overall already reasonably capable of testing a file system.

## 6.1 Future Work

**Fine Granular Testing** During our tests, we experienced that WineFS provides stronger crash consistency guarantees than can be easily tested with Vinter. Specifically, in strict mode, each file system operation is synchronous. This semantic is translated to a Vinter test by inserting a checkpoint between each file system operation. However, Vinter tests are usually a series of shell commands. Each shell command executes multiple system calls, often to open, modify, and close the file. Vinter cannot insert a checkpoint in the middle of a command. Therefore, such Vinter tests cannot determine if a single operation is synchronous.

To circumvent this issue, we wrote custom programs that explicitly called each required system call and inserted a checkpoint in between. While this works for few test cases, it is currently complex to create new or modify existing test cases.

It might be possible to automatically detect relevant system calls and insert checkpoints between them to better match the provided crash consistency guarantees. Alternatively, Vinter test cases could feature an advanced set of commands that allows specifying tests at the same granularity as system calls.

**Vinter Performance** In Section 5.11 we described how we improved the performance of Vinter significantly. However, further optimizations might be possible.

Only tests for the same VM file can be executed in parallel using the same Vinter instance. When executing tests for different VM files, the test runs must be executed sequentially. This could be improved to either execute each tests for a list of VMs or by passing tuples of test files and VM files to Vinter.

When executing only few tests, it might be interesting to further parallelize the execution of each individual test. This could be achieved by generating crash states in parallel. In our tests, this phase made up between 21% and 71% of each Vinter invocation while using parallel semantic state extraction. Specifically, long Vinter runs spend a big portion in this phase. Performance improvements are most needed for long Vinter runs. This can primarily be achieved by optimizing the crash state generation phase.

**Automated Result Analysis** Vinter is good at finding potentially erroneous crash states and testing for properties like atomicity and single final state. An atomic and single final state consistent operation is usually fully correct in terms of crash consistency. However, beyond these properties, there is no mechanism to specify which crash states are at all legal for a given test.

Manually deciding if a crash state is valid is very error prone. If Vinter was able to decide if a crash state is invalid, the workload on the user could be reduce significantly. A simple approach would be to specify valid crash states in each Vinter test.

Chipmunk solves this issue using an oracle that automatically generates valid file system states. "The oracle runs the original workload on a fresh file system instance in parallel with log replay." [30] Such a mechanism would also be useful for Vinter. Note that such a system must be adapted to the respective crash consistency guarantees of each file system. For example, WineFS in relaxed mode may only partially execute a write system call while only the initial and completed state are valid in strict mode.

# Bibliography

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/kernel/gcov/gcc_4_7.c. Accessed: 2023-08-15.

[2] Making filesystems exportable. https://docs.kernel.org/filesystems/nfs/exporting.html. Accessed: 2023-09-26.

[3] std::process - rust. https://doc.rust-lang.org/std/process/index.html. Accessed: 2023-09-26.

[4] Using gcov with the linux kernel. https://www.kernel.org/doc/html/v4.14/dev-tools/gcov.html. Accessed: 2023-08-15.

[5] Persistent memory FAQ. https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/persistent-memory-faq.html, 2 2020. Accessed: 2023-05-13.

[6] 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, 1 2021. Accessed: 2023-05-13.

[7] Memory tiering: A new approach to solving modern data challenges. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/memory-tiering-improving-data-management-paper.html, 4 2022. Accessed: 2023-05-13.

[8] GNU coreutils 9.3. https://www.gnu.org/software/coreutils/manual/coreutils.html, 2023. Accessed: 2023-08-15.

[9] GNU make. https://www.gnu.org/software/make/manual/make.html, 2 2023. Accessed: 2023-08-03.

[10] Intel® 64 and IA-32 architectures software developer's manual. https://cdrdv2.intel.com/v1/dl/getContent/671200, 3 2023. Accessed: 2023-05-23.

[11] Intel® C++ compiler classic developer guide and reference. https://cdrdv2-public.intel.com/767250/cpp-compiler_developer-guide-reference_2021.8-767249-767250.pdf, 3 2023. Accessed: 2023-05-14.

[12] Linux man-pages - manual pages for GNU/Linux. https://mirrors.edge.kernel.org/pub/linux/docs/man-pages/man-pages-6.05.tar.xz, 8 2023. Accessed: 2023-08-02.

[13] QEMU user documentation. https://www.qemu.org/docs/master/system/qemu-manpage.html, 2023. Accessed: 2023-08-08.

[14] Using the GNU compiler collection. https://gcc.gnu.org/onlinedocs/gcc.pdf, 2023. Accessed: 2023-08-15.

[15] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, 8 2018.

[16] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2592798.2592814.

[17] Bryan Henderson. hwclock(8) manual page. https://github.com/util-linux/util-linux/blob/ab7fe95ad7495adac41a4d79f4771c1b4cbe1fc0/sys-utils/hwclock.8.adoc, 6 2023.

[18] Rohan Kadedodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Greg Ganger, Aasheesh Kolli, and Vijay Chidambaram. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, 10 2021.

[19] Rohan Kadekodi. Issue 1 fix. https://github.com/utsaslab/WineFS/pull/7/, 12 2021. Accessed: 2023-08-02.

[20] Rohan Kadekodi. Issue 3 fix #8. https://github.com/utsaslab/WineFS/pull/8/, 12 2021. Accessed: 2023-06-01.

[21] Rohan Kadekodi. [wip] issue 5 fix. https://github.com/utsaslab/WineFS/pull/6/, 12 2021. Accessed: 2023-06-01.

[22] Rohan Kadekodi. Winefs source code. https://github.com/utsaslab/WineFS/tree/b4017d0fa5fd2b526e870b0338c311829e5f4464/Linux-5.1/fs/winefs, 1 2022. Accessed: 2023-06-06.

[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, Carlsbad, CA, July 2022. USENIX Association. https://www.usenix.org/conference/atc22/presentation/werling.

[24] Rob Landley. Ramfs, rootfs and initramfs. https://www.kernel.org/doc/html/latest/filesystems/ramfs-rootfs-initramfs.html, 10 2005. Accessed: 2023-08-15.

[25] Hayley LeBlanc. Possible crash consistency issue with truncate using multiple file descriptors · issue #9 · utsaslab/winefs · github. https://github.com/utsaslab/WineFS/issues/9, 1 2022. Accessed: 2023-08-31.

[26] Hayley LeBlanc and Rohan Kadekodi. Possible crash consistency issue with write · issue #3 · utsaslab/winefs · github. https://github.com/utsaslab/WineFS/issues/3, 12 2021. Accessed: 2023-08-31.

[27] Hayley LeBlanc and Rohan Kadekodi. Write may not be atomic with respect to crashes in strict mode · issue #5 · utsaslab/winefs · github. https://github.com/utsaslab/WineFS/issues/5, 12 2021. Accessed: 2023-08-31.

[28] Hayley LeBlanc and Rohan Kadekodi. Possible crash consistency bug with write() · issue #1 · utsaslab/winefs · github. https://github.com/utsaslab/WineFS/issues/1, 1 2022. Accessed: 2023-08-31.

[29] Hayley LeBlanc, Rohan Kadekodi, and YozoraHoshifuru. Issues · utsaslab/WineFS · GitHub. `https://github.com/utsaslab/WineFS/issues?q=is%3Aissue`, 1 2022. Accessed: 2023-06-01.

[30] Hayley LeBlanc, Shankara Pailoor, Om Saran K. R. E., Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems, 5 2023.

[31] David Mulnix, Arijit Biswas, Ruchira Sasanka, Vinodh Gopal, Wajdi Feghali, Mahesh Wagh, Alberto Villarreal, and Dan Zimmerman. Technical overview of the 4th gen intel® xeon® scalable processor family. `https://www.intel.com/content/www/us/en/developer/articles/technical/fourth-generation-xeon-scalable-family-overview.html`, 7 2022. Accessed: 2023-08-03.

[32] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-X86 architecture. *Proc. ACM Program. Lang.*, 4(POPL), 12 2019. `https://doi.org/10.1145/3371079`.

[33] Andy Rudoff. 8 byte atomicity & larger store operations. `https://groups.google.com/g/pmem/c/6_5daOuEI00/m/rEJnjKzCCAAJ`, 11 2020. Accessed: 2023-08-03.

[34] Steve Scargall. *Programming Persistent Memory*. Apress Berkeley, CA, 1 2020.

[35] Linus Torvalds. Linux 5.1. `https://lkml.org/lkml/2019/5/5/278`, 5 2019. Accessed: 2023-08-15.

[36] Pantea Zardoshti, Michael Spear, Aida Vosoughi, and Garret Swart. Understanding and improving persistent transactions on optaneâ„¢ dc memory. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 348–357, 2020.