Karlsruhe Institute of Technology

Fast Persistent Memory Crash Consistency Analysis based on Virtual Machines

Bachelor's Thesis
submitted by

Thomas-Christian Oder

to the KIT Department of Informatics

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

3. July 2023 – 3. November 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, November 3, 2023

Abstract

Non-volatile memory (NVM) is a new technology that is directly integrated in the processor's memory system. While NVM has a higher latency and is slower than DRAM, its performance is far superior when compared to regular SSDs. NVM can be used as a DRAM replacement as well as regular storage. When used as regular storage, it can be accessed by a Persistent Memory (PM) compatible application, or be abstracted into a regular storage device with a generic standard interface, like POSIX, using a file system. New file systems like `Nova` and `PMFS` were developed to fully profit from NVM's design. Due to this difference in design, existing file system testers to test for crash consistency, like CrashMonkey or Hydra, are not able to test those file systems and new solutions, like Vinter, had to be developed.

Vinter is a record-and-replay black-box approach for testing PM file systems using manually written tests. While it provides a quick test heuristic already, Mumak, a black-box system to analyze performance and crash consistency on PM applications, presented a different approach that promises similar results at a better runtime performance when applied to Vinter. Mumak uses a trace entry's kernel stack trace to generate a *failure point tree*. This tree is used deduplicate trace entries by their kernel stack trace and to generate possible crash images later on. Mumak extends this process with a pattern based trace analyzer to discover potential bugs in performance and design of the tested application.

In this thesis, we extend Vinter's crash image generator by this new approach to generate crash images using the *failure point tree*.

We also extend Vinter with an improved and standalone version of Mumak's trace analyzer to provide an additional way to find bugs.

We could verify that the *failure point tree* approach delivers a big improvement in runtime while delivering similar, although not always fully equal, results when running our existing tests against `Nova`, `Nova-Protection` and `PMFS`.

We were further able to show that the trace analyzer is able to find some of the bugs missed by the *failure point tree* approach as well as additional bugs that were not previously covered at all.

Contents

Abstract **v**

Contents **1**

1 Introduction **3**

2 Foundations **5**
2.1 Persistent Memory 5
2.2 Crash Consistency 7
2.3 Vinter 8
2.4 Chipmunk 8
2.5 Mumak 9
2.6 YAT 10

3 Design **11**
3.1 Overview 12
3.2 Crash Image Generation 13
3.2.1 Failure Point Tree 14
3.3 Trace Analysis 16

4 Implementation **19**
4.1 Crash Image Generation 20
4.1.1 Failure Point Tree 21
4.2 Trace Analysis 23
4.3 Additional Changes 28
4.3.1 Documentation and Scripts 28
4.3.2 Analysis 29
4.3.3 Tools 29

5 Evaluation **31**
5.1 Test Setup 31

5.2 Crash Image Generation 33
5.2.1 Runtime Performance 33
5.2.2 Result Quality 40
5.3 Trace Analyzer 45
5.3.1 Runtime 45
5.3.2 Quality 47

6 Discussion 49
6.1 Failure Point Tree 49
6.2 Trace Analyzer 50

7 Conclusion 53
7.1 Future Works 54

Glossary 54

A Test Performance Summaries 59
A.1 Crash Image Analysis 60
A.2 Trace Analysis 68

References 71

Chapter 1

Introduction

Non-volatile memory (NVM) is a new byte-addressable storage technology. Its first and most popular commercially available implementation is the Intel® Optane™ DC Persistent Memory Module (Optane DC PMM) [9]. Optane DC PMM is a nonvolatile memory DIMM that is able to work as RAM replacement at a better price to storage performance, or as a non-volatile storage directly addressable as part of the processor's memory hierarchy.

The latter is implemented by either directly accessing the Persistent Memory (PM) through software or through a PM compatible file system providing common user space interfaces such as POSIX [7].

For software to directly access the PM, the software will be required to implement all data-, storage- and crash-handling on its own, requiring its entire design to be around PM.

File systems designed for PM on the other hand require no changes in an application while still providing the advantages delivered by PM. An ideal file system provides performance while guaranteeing crash consistency. Crash consistency is an application or file system's ability to recover from a crash consistently, leaving no inconsistent or incomplete data behind.

The rather young age of the NVM technology paired with its difference in design make PM file system development hard. As such, proper testing is important.

Previous tools to test for crash consistency, like CrashMonkey paired with ACE [19] or Hydra [14], rely on the kernel's block layer to record I/O operations for their analysis. This makes them incompatible with PM file systems.

To properly test PM file systems, new tools were introduced.

Vinter [11] uses binary translation to trace relevant instructions, like load-store instructions and barriers, in a virtual machine to provide a black-box debugging approach without the need for kernel or user space code modifications. Given the amount of possible crash images during a test, analyzing all of them would not be feasible because it would take too long. Vinter uses a heuristic to only analyze crash images that contain variations of stores that are likely to be read during the post-failures crash-recovery stage. Although this limits the amount of crash images to check, its performance could still be improved. LeBlanc et al. [18] have observed that logic bugs are much more common in PM file systems than programming errors. This is where Mumak comes into play.

Mumak provides performance and crash consistency analysis for PM applications by analyzing the software binary with Intel Pin [23] to quickly generate a *failure point tree* that is used to deduplicate the trace entries, and therefore their resulting crash images, by their kernel stack trace. This is paired with a comprehensive *trace analysis* to find certain types of PM misuse.

The *failure point tree* promises similar results in crash consistency testing while requiring less crash images to be generated. This will greatly improve a test's runtime because fewer crash images need to be generated and processed.

To further improve on detecting potential performance, design and consistency bugs, Mumak presents a trace analysis tool that detects potential bugs using pattern matching on a test's trace.

In this thesis we will focus on implementing Mumak's *failure point tree* and an improved version of its *trace analysis* into Vinter's test pipeline to enable an even faster crash image generation and a comprehensive analysis of logic bugs in PM file systems. To verify the implementation and to evaluate the bug hit rate we will compare the result to the original implementation of Vinter in terms of performance, runtime and the amount of bugs that were found. The source code of the changes described in this thesis can be found on GitHub.com/Myself5/vinter [22].

Chapter 2

Foundations

2.1 Persistent Memory

Persistent Memory (PM) is a type of non-volatile memory (NVM). NVM is a new byte addressable storage technology integrated in the processor's memory system. It allows software direct access to virtual memory mappings in order to persistently load and store data.

PM's first commercially available implementation is Intel® Optane™ DC Persistent Memory Module (Optane DC PMM). Optane DC PMM is a DDR4 DIMM module using the Optane DC PMM UTH DDR-T Protocol [8]. Because of that, its compatibility is limited to Intel CPU's which support the DDR-T protocol.

Optane DC PMM's performance sits between SSDs and DRAM with a higher latency than DRAM (346 ns) [9] but, especially when interleaved and used with an optimal amount of threads, a far superior read- (40 GB/s) and write-bandwidth (10 GB/s) compared to SSDs [27].

As Optane DC PMM is both byte-addressable and persistent, it can be used as a DRAM replacement with high capacity and slightly higher latency for main memory (memory mode) or as a persistent storage device replacing disks and SSDs (AppDirect mode).

In this thesis, we will focus on its usage in AppDirect mode. In AppDirect mode, software is able to directly access the storage through the processor's memory hierarchy. For software to access the PM, it is required to manually handle all storage processes like implementing memory fences, flushing volatile caches as well as handling crashes and crash recovery, resulting in the software to be written entirely for and around the design of Persistent Memory.

Alternatively, new file systems were designed around PM to provide generic standards (our focus will be on POSIX compatible file systems during the tests), abstracting the PM to the software while profiting from the performance improvements delivered by PM. However, these file systems still need to handle the aforementioned challenges and require testing. Due to the differences in PM's design, software and file systems can no longer be tested with previously known analysis tools like CrashMonkey paired with ACE [19] or Hydra [14] as those rely on the kernel's block layer to analyze I/O operations and only inject crashes after `fsync`-related system calls. PM file systems do not use `fsync` but instead require a set of cache line flushes (`clflushopt`) or write backs (`clwb`) paired with memory fences to persistently store the data onto the NVM. As such, it is necessary to also inject crashes before, during and after these system calls to expose bugs in the largely untested and complex crash consistency mechanisms of PM file systems. Further, LeBlanc et al. [18] observed that logic bugs are much more common than actual programming errors on PM file systems. To handle this change in requirement, new tools were introduced to debug PM file systems and applications.

2.2 Crash Consistency

Crash consistency is an often only loosely described property of a stateful application. It describes a storage's ability to consistently store persistent data when experiencing a crash. When data is written to NVM it is initially written to a *volatile* cache (often DRAM) and will be stored to the persistent NVM upon an instructed memory write back or flush. If a crash happens during that procedure the cache is either lost, partially stored or fully stored to the NVM. We call these different states *crash images*. For an *atomic* operation this means that each crash image results in one of two states: the initial or the final state. Figure 1 shows the NVM storage process during an atomic operation as time proceeds from left to right.

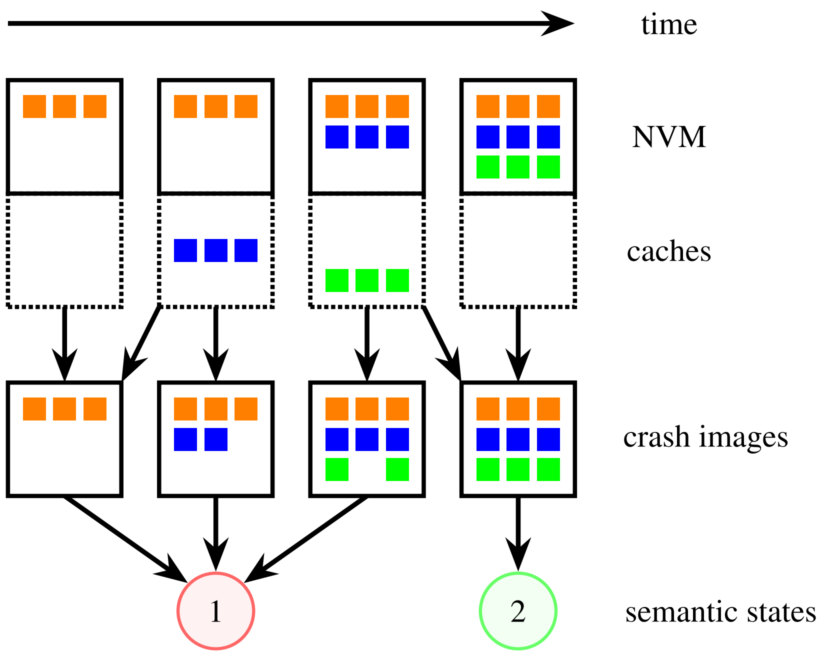


Figure 1: Potential crash images recovering to one of two semantic states after an atomic operation [11]

When the write of new data (here marked blue) is instructed, the data is sent to a cache and as long as these caches are not fully flushed to the NVM, like they are in the final step on the right, the recovery operation ends up in the same, initial semantic state. The final write (here marked green) marks a successful write, comparable to how journaling works on some file systems. For non-atomic operations Kalbfleisch, Werling, and Bellosa define a *single final state (SFS)* with weaker properties that all crash images recover to. Any of these properties might however be violated by different kinds of logic or programming bugs. To find these bugs, various testing frameworks of different scopes were introduced or are under active development.

2.3 Vinter

Vinter [11], the virtualization-based NVM tester, is a record-and-replay black-box approach for testing a PM file system's crash consistency. It uses manually written tests to cover most of the POSIX standard but also provides guidance on how to extend those tests if needed. As its name suggests, Vinter uses binary translation to trace relevant load-store instructions and barriers in a virtual machine using PANDA [1]. Using that trace, *crash images* representing possible NVM contents after a crash, are generated. These *crash images* are reduced exponentially in quantity by using a heuristic to identify the NVM locations where the file system's recovery code is reading from the crash image. This is followed by comparing each *crash image's* semantic state, in this case a list of all files in a file system, to automatically verify crash consistency properties such as atomicity.

2.4 Chipmunk

Chipmunk [18] is a record-and-replay grey-box approach for testing a PM file system's crash consistency using the ACE workload generator [19] and Google's syzcaller grey-box kernel fuzzer [25] to generate test workloads. It requires only basic knowledge about the file system and needs to be supplied with the name (for kernel space components) or an offset (for user space components) of a file system's persistence functions so that it can instrument them at runtime using Kprobes [13] for kernel space components, Uprobes [26] for userspace components or both in case of a split file system like SplitFS [10]. Other than that, it does not require changes to the software, kernel or file system allowing easy debugging without introducing an additional layer of potential errors. [18]

2.5 Mumak

Mumak is a black-box system to analyze performance and crash consistency in PM applications. Other than Vinter and Chipmunk, it focuses on PM applications rather than PM file systems. Like Vinter, Mumak relies on a given workload to run the analysis on an application and is therefore only able to test what is covered in the given workload. Instead of filtering all possible crash images, Mumak generates a *failure point tree* by using the Intel Pin tool to capture opcodes and optional arguments. Specifically, Mumak captures stores, flushes, fences and atomic instructions that have a fence semantic to find potential failure points and uses these to generate the *failure point tree*. An example of such a *failure point tree* and its corresponding code is shown in Figure 2.

```c
1  void persist(int *i) {
2      clwb(i);
3      sfence();
4  }
5  void redundant_loop(int *i) {
6      for (int j = 0; j < 8; j++)
       {
7          *i = 3;
8          persist(i);
9      }
10 }
11 int main() {
12     int *x = pm_alloc();
13     int *y = pm_alloc();
14     int *z = pm_alloc();
15     *x = 1;
16     clwb(x);
17     sfence();
18     *y = 2;
19     persist(y);
20     redundant_loop(z);
21     return 0;
22 }
```

pm.c

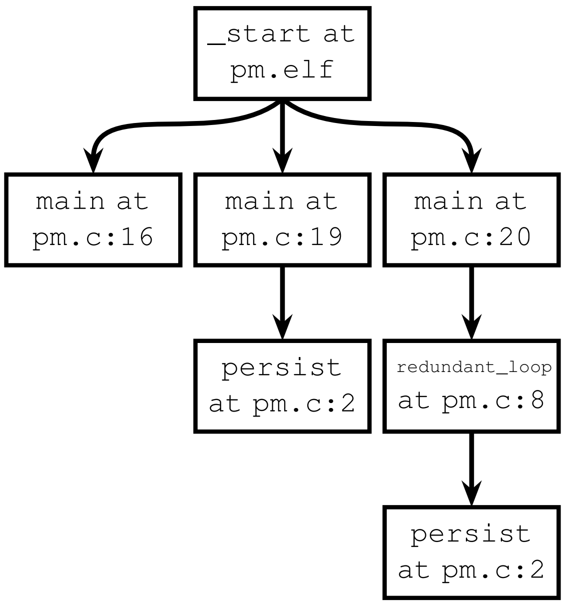


Figure 2: Sample program and corresponding failure point tree [3]

Using that tree, Mumak iterates over the tree until it reaches an unvisited failure point in the tree, injects a fault and marks the point as visited. It then recovers the state and, depending on the recovery status, reports a discovered bug. This process is repeated until all leaves in the *failure point tree* are marked as visited. To cover potential bugs not reached by the tree, Mumak generates a PM access trace while processing the failure point tree and analyzes that trace to find patterns of misuse to generate bug reports or warnings in parallel.

The following PM patterns are being observed: [3]

- Store instructions that are not explicitly persisted.
- Flush instructions that act on volatile address(es), or whichs address(es) have not been written to the cache since the most recent flush.
- Fence instructions without pending flush or non-temporal stores.
- Fence instructions that act on more than one `clflushopt`, `clwb`, or non-temporal store.

If a given pattern is found in the trace, Mumak will report it. For performance reasons, this will happen after a single pass. Due to that, paired with the black-box approach, Mumak may not always be able to confidently say if it is in the presence of a bug and will only report a warning to the developer in those cases. The result of the *failure point tree* processing and the trace analysis form the final report provided by Mumak.

2.6 YAT

YAT [15] is a hypervisor-based brute-force testing tool designed for Intel's PMFS file system [2]. It records individual memory states without filtering or evaluation on how useful a specific crash state might be. It was one of the first PM file system testing frameworks and shows how important it is to slim down the crash image generation. The authors report that, due to the lack of filtering, going through all possible crash images generated in one of their three test workloads would take over 5 years to be fully completed.

Chapter 3

Design

With Vinter's test concept leaving room for performance improvement and the knowledge that most of the bugs in a PM file system are logical bugs, we intend to extend Vinter with a different approach to generate crash images as well as a trace analysis tool in this work.

Our primary objective is to greatly improve Vinter's runtime while maintaining a similar quality in results. This means that, between the existing heuristic and the new approach, the tests should ideally have the same amount and kind of semantic states as their result while maintaining a shorter runtime.

With the *failure point tree* approach we will skip the heuristic to create additional crash images that could likely be of interest entirely. The approach will only try to generate up to two crash images for every `flush` or `fence` that had a previous `write` to it. In addition, we implement and use the *failure point tree* during the crash image generation to deduplicate the trace entries for which a crash image should be generated by their kernel stack trace. This prevents creating multiple crash images for the same kernel stack trace.

If a crash image was generated for a trace entry with the same stack trace already, there will be no additional crash image generated, and the next trace entry will be processed.

Using that approach, we intend to achieve a faster runtime because, ideally, less crash images will need to be generated and analyzed to discover the same amount of semantic states.

By introducing trace analysis we will get an additional way to discover performance and durability bugs in a given file system. While the discovery of performance bugs is a new feature, the additional discovery of durabilty bugs specifically complements the *failure point tree* based crash image generation as Gonçalves, Matos, and Rodrigues describe this ability to be the *failure point tree* crash image generator's weakness in their paper [3].

3.1 Overview

A regular Vinter test run consists of the three steps shown in Figure 1.

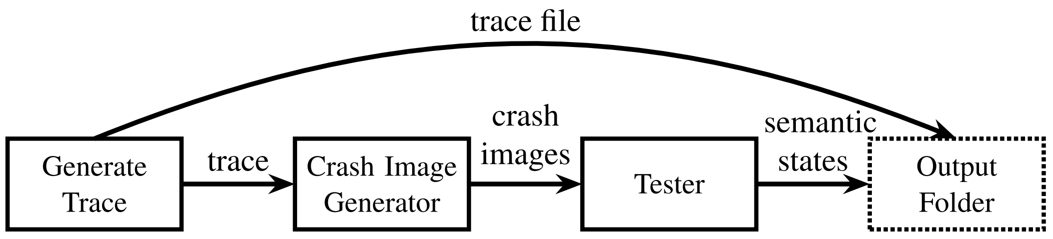


Figure 1: original Vinter flowchart

In a full test run, Vinter runs a given test and generates a trace. The existing tests are rather simple and can be as easy as adding content to a file in `test_append`.

The trace is then stored in the output folder and the path will be given to the crash image generator. The crash image generator proceeds to parse the trace, generates crash images that could be of interest and passes them to the tester. A crash image contains the program in a specific state during which a crash is injected. The tester observes the file system's internal recovery from said crash and logs the recovery's output or a failure if the recovery fails. The tester's *state extractor* proceeds to mount the file system in a read-only state and provides a serialized representation of each file. This serialization is used to determine differences in files. A crash image's output that differs from the expected output we would get if there were no crashes during the run, called *single final state*, will be reported as an additional *semantic state* and is considered a bug in the file system.

In this work, we extend the crash image generator with a different approach to create crash images, as detailed in Section 3.2.

With potential degradations in bug-detection quality in mind, we will extend Vinter by a trace analysis tool that takes a given trace file and checks it for potential design and performance bugs by analyzing the order of trace entries and operation range. The adjusted flowchart for this, now four-step, process is shown in Figure 2, and the trace analyzer is detailed in Section 3.3.

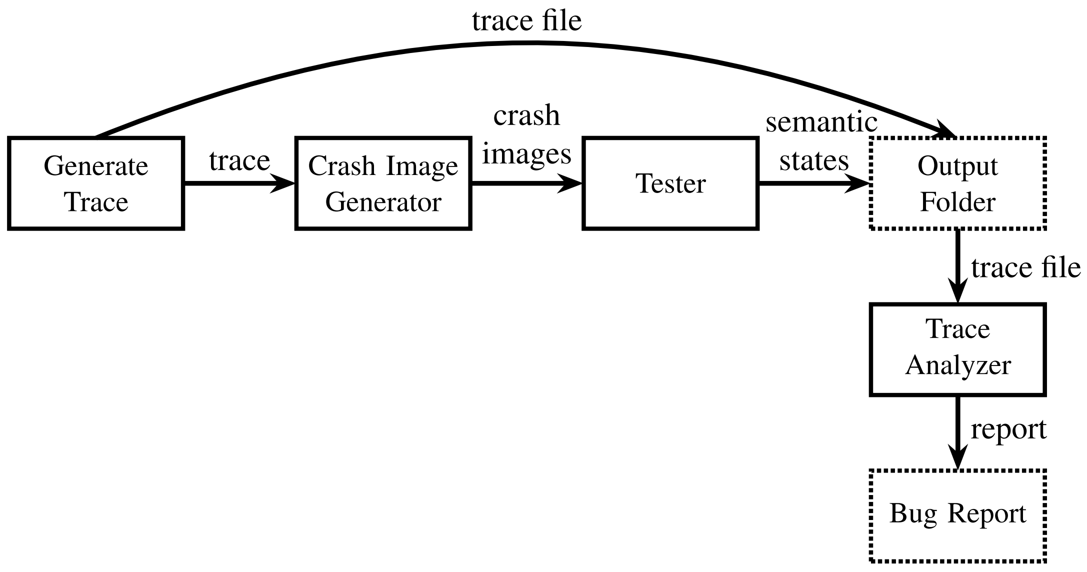


Figure 2: Vinter flowchart with added trace analysis

3.2 Crash Image Generation

A trace generated by Vinter logs all writes, fences, flushes and hypercalls during a given test run. As the trace includes the startup and shutdown of the test VM, hypercalls are used to determine the beginning, the end and the individual checkpoint steps of a test run. The crash image generator begins to iterate over the given trace and proceeds to filter the entries within the test range. The filter works by determining if there are currently unstored writes, that are followed by a fence instruction within the same checkpoint segment. In that case the trace entry will be sent off to generate crash images using the heuristic.

To improve Vinter's runtime while keeping the same quality in result, we try to reduce the number of similar crash images that need to be generated and analyzed. To do so, we extended the aforementioned trace filter to act the same for flushes and fences and introduced a *failure point tree*. This tree is used to deduplicate the trace entries that could result in the generation of crash images by their kernel stack trace. It does so by attempting to insert the kernel stack trace as detailed in Section 3.2.1. During insertion, the *failure point tree* checks if a stack trace is already included in the tree. If the stack trace is fully included already, no additional crash images will be generated. If not, the stack trace will be added to the tree and crash images will be generated.

The *failure point tree* implementation does not use the heuristic to generate additional crash images that could be of interest and only generates a single crash image without pending writes and, if present, one including pending writes per trace entry.

3.2.1 Failure Point Tree

In order to prevent creating multiple crash images for the same code at different positions in the trace we use the entry's kernel stack trace and a *failure point tree*. When trying to insert the stack trace into the *failure point tree*, the tree either returns true, if successfully inserted, or false, if the entry is already present in the tree.

Table 3.1: Snippet from a trace running `test_rename-dir` with `vm_nova`. `pc` and `kernel_stacktrace` were altered and shortened for a better readability.

Trace Entry	**Trace ID**	**Stack Trace**
`Write`	`295921`	`[19, 23, 25, 35, 39, 48]`
`Flush`	`295922`	`[25, 35, 39, 48]`
`Write`	`295924`	`[25, 35, 39, 48]`
`Flush`	`295925`	`[25, 35, 39, 48]`
`Write`	`295934`	`[35, 39, 48]`
`Flush`	`295935`	`[35, 39, 48]`
`Write`	`296056`	`[25, 28, 48]`
`Flush`	`296057`	`[25, 28, 48]`

Table 3.1 shows a trace snippet containing four entry pairs that match Vinter's filter requirements by having a `write` followed by a `flush` or `fence`. Starting with an empty *failure point tree*, the `flush` with ID `295922` will be added to the tree and the corresponding crash images will be generated. The following `flush` with ID `295925` matches the filter as well. However, as the stack trace is the same as the previously added entry, the crash image generation is skipped, and no new crash images will be generated. The trace entries with ID `295935` and `296057` have a different stack trace that are not contained in the *failure point tree* yet. Therefore, their stack trace will be added to the *failure point tree* and crash images will be generated. To get an overview on how this tree would be structured, a tree matching the snippet in Table 3.1 is shown in Figure 3. The address zero is used as a common root to span the tree.

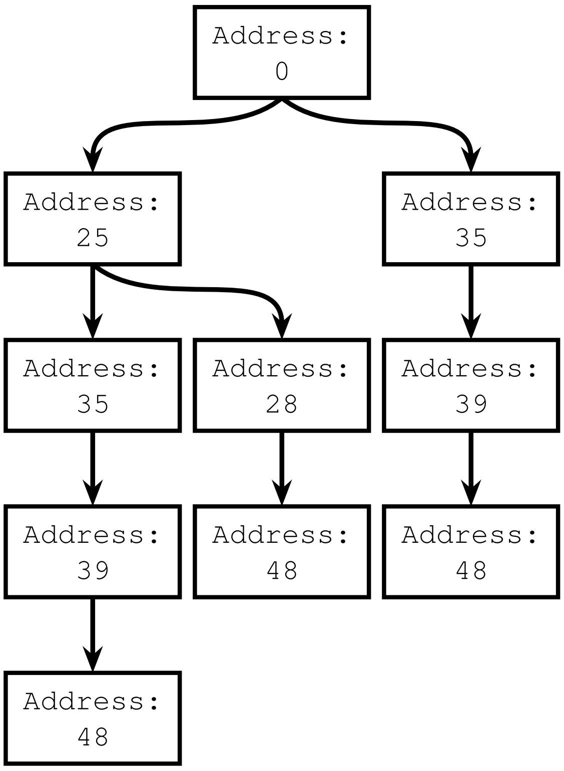


Figure 3: *Failure point tree* resulting from trace snippet shown in Table 3.1

3.3 Trace Analysis

The standalone trace analysis is intended to parse an existing trace file and use it to find potential bugs by analyzing the order and address space of trace entries to determine possible overlaps. In its original design, Mumak differentiates between *correctness bugs*, which more specifically get split up in *durability* and *ordering bugs*, and *performance bugs*. We adapt that differentiation alongside the seven possible bug types listed and detailed in Table 3.2.

Table 3.2: List of possible bugs during trace analysis

Bug	**Type**	**Description**
Redundant Flush	Performance	Flushes are considered redundant when the content of the address they are supposed to store has not been altered since the address has last been flushed, if the flush acts on volatile memory, or if the only writes were non-temporal and went straight to storage, skipping the cache.
Redundant Fence	Performance	Fences are considered redundant if there were no flushes or non-temporal writes to the address since the last fence.
Missing Flush Missing Fence	Durability	Stores that are not explicitly flushed and fenced and therefore rely on non-deterministic cache eviction policies are reported to have a missing flush or fence.
Overwritten Unflushed Overwritten Unfenced	Durability	Overwriting an address without previously fencing/flushing it implies the address to be used as a volatile memory of sorts. This should be moved to actual volatile memory and is therefore reported as a bug.
Unordered Flushes	Ordering	When storing data that requires multiple flush instructions the flush instructions should be in order to allow a successful recovery from a crash. A subset of this bug could also be considered as an *atomicity bug* as, ideally, a set of multiple flushes should be performed atomically.

Continued on next page

Table 3.2 – continued from previous page

Bug	Type	Description
Implicit Flush	-	Mumak contains code for this, but it was commented out. Mumak's paper has no documentation on this decision or that bug in general, so we did not implement its functionality. [5, 6]

Performance bugs are bugs that negatively affect the performance by running instructions unnecessarily whereas *correctness bugs* list bugs that could potentially reveal mistakes in a file systems design due to incorrect usage or *consistency* bugs caused by incorrect *ordering* that could result in an incorrect crash recovery.

Table 3.3 shows two trace entries that will result in a bug.

Table 3.3: Trace example for an Overwritten Unflushed bug

Trace Entry	Trace ID	Address
`Write`	`295752`	`2990248`
`Write`	`295808`	`2990248`

The trace analysis iterates over a given trace and checks the order as well as the address range of each trace entry. If an address range is affected by a given pattern of trace operations that matches a known bug, in this case two writes to the same address without a fence or a flush in between, a bug in that entry will be reported. For Table 3.3 that is an overwritten unflushed bug for ID `295752`. As those bugs could happen multiple times in the same trace, the list of bugs will be deduplicated by their stack trace, type and checkpoint within the trace using a modified *failure point tree*.

To do so, we extended the *failure point tree's* leaves to contain a list of bugs assigned to this specific leaf. As the leaf structure is created from the kernel stack trace, this will assign the bug to its responsible trace entry's kernel stack trace. We further extended the tree's code to optionally allow inserting a new bug alongside the kernel stack trace and to check the list of bugs during insertion to maintain the same return handling.

Chapter 4

Implementation

To improve Vinter's overall runtime, we implemented a new approach using a *failure point tree* to deduplicate trace entries and their resulting crash images during crash image generation.

To find and report potential *performance* and *correctness* bugs in a given trace, we introduced and implemented a trace analyzer. The source code to the changes described in this section can be found on GitHub.com/Myself5/vinter [22].

Vinter's Rust implementation is split into four different Cargo packages.

- `vinter_common`: the library for common code between the different tools
- `vinter_trace`: the library responsible for creating a trace
- `vinter_report`: the standalone tool to print a trace file
- `vinter_trace2img`: the tool to run a full test using `vinter_trace` to generate a trace followed by creating crash images, and determining the different semantic states when recovering from those crash images

In Section 4.1 we describe the extension of `vinter_trace2img` with the *failure point tree*, it's implementation and how it was implemented into the crash image generator alongside the existing heuristic.

`vinter_report`'s extension with the pattern based trace analyzer, its way of working and the improvements compared to Mumak's original design are outlined in Section 4.2.

In addition, we document the changes to prepare for the trace analyzer and changes that prove themselves to be useful for either testing or evaluation in Section 4.3. This section also documents our changes to the documentation, the new tests we introduced and our changes to existing test tools that were required for them to work with the *failure point tree* implementation.

4.1 Crash Image Generation

By default, Vinter's crash image generation step uses the `HeuristicCrashImageGenerator`. As its name implies, this generator handles the creation of crash images by parsing a trace and applying the heuristic to generate additional crash images that could likely be of interest to find varying semantic states. The previous implementation already contained a hard coded boolean to toggle between using the heuristic and using no heuristic, which would result in generating every possible crash image. Moving this hard coded switch to a CLI flag had been an outstanding to-do already, so the to-do was implemented as `-g/--generator <(d)efault|(f)pt|(n)one>` and extended to become an internal parameter that is set when creating a new `HeuristicCrashImageGenerator` instance. Furthermore, the class was renamed into `GenericCrashImageGenerator` to reflect its new features.

The generator parses a previously generated trace in the `replay` method. This method start off by finding the test range using the hypercall trace entries and proceed to parse the entries within that range. Essentially, the `replay` method calls the generation of crash images for every `hypercall` and every `fence` if there was a previous `write` that has not been followed up by a `fence` yet.

We extended this `replay` method to handle a `flush` the same way it handles a `fence` when using the *failure point tree* implementation and made it forward the trace entry's kernel stack trace from its metadata to `insert_crash_image`.

The `insert_crash_image` is the code responsible for creating a crash image. It receives a trace entry and proceeds to generate a crash image without pending writes and, if present, including pending writes. It will then proceed to apply the heuristic the find and create additional crash images that will likely be of interest.

We extended this method with a `callstack_option` parameter that includes an option to an entry's kernel stack trace. The way `replay` is configured, this option will only be `None` if the trace entry is a *hypercall*. In that case, we will skip the *failure point tree* and always generate a crash image. Otherwise, the stack trace is of the type `Some(Vec<u64>)`. It will be unwrapped, preceded with a zero to introduce a common root that is required for the tree structure and will be attempted to be inserted into the tree.

In case of successful insertion, crash images without pending writes and, if present, including pending writes will be generated. All additional crash image generation will be skipped for the *failure point tree* based test run.

4.1.1 Failure Point Tree

We modeled the initial *failure point tree* after Mumak's implementation. After research, the only existing library that provides support for non-binary trees that is still in active development is `r3bl_rs_utils` [24]. However, even the code examples provided by the developer were broken at the time of implementation. As the library, even if it was working, did not provide all the search features required by the initial design and as the library-provided multi-thread safety was not a concern in our implementation, we decided to implement the tree on our own.

The *failure point tree* has a single root leaf that, by itself, has a vector of links to its children. In our implementation, the root leaf will be represented by the address `0`. Each child has a parent reference to its single parent and a vector containing a reference to its children.

Due to the difficult ownership handling between children and parent, we followed an example on double linked lists from *Learning Rust With Entirely Too Many Linked Lists* [17] and introduced the `FPTraceLink: Option<NonNull<FPTraceAddr>>` type. While this requires the usage of `unsafe` when unwrapping the `FPTraceLink`, the *failure point tree* provides a controlled environment that guards the `unsafe` operations safely.

Due to a misunderstanding, the initial design contained the ability to optionally store a `CrashImageHash` in each `FPTraceAddr`. This was caused by erroneously assuming the `fence_id` in a crash image entry's `originating_crashes` vector to be the stack trace address and assuming that all `fence_id`s in the vector would make up the entry's stack trace.

After clearing up that confusion, we extended PANDA's CLI parameter to collect a kernel stack trace as part of the metadata when using the *failure point tree*, the `CrashImageHash` support was removed from the tree, and a proper kernel stack trace handling had been put in place by getting the kernel stack trace from a trace entries `metadata` and forwarding it to `insert_crash_image` inside the `replay` method.

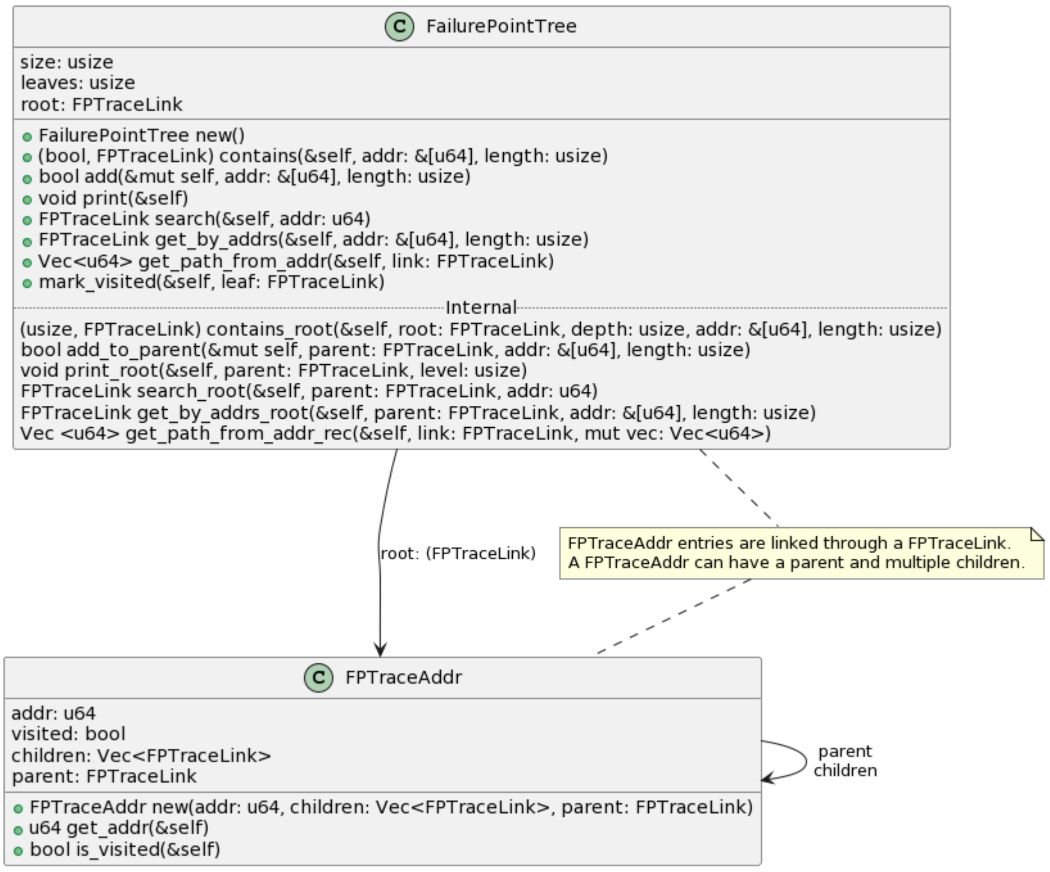

FailurePointTree
size: usize
leaves: usize
root: FPTraceLink
FailurePointTree new()
(bool, FPTraceLink) contains(&self, addr: &[u64], length: usize)
bool add(&mut self, addr: &[u64], length: usize)
void print(&self)
FPTraceLink search(&self, addr: u64)
FPTraceLink get_by_addrs(&self, addr: &[u64], length: usize)
Vec<u64> get_path_from_addr(&self, link: FPTraceLink)
mark_visited(&self, leaf: FPTraceLink)
Internal
(usize, FPTraceLink) contains_root(&self, root: FPTraceLink, depth: usize, addr: &[u64], length: usize)
bool add_to_parent(&mut self, parent: FPTraceLink, addr: &[u64], length: usize)
void print_root(&self, parent: FPTraceLink, level: usize)
FPTraceLink search_root(&self, parent: FPTraceLink, addr: u64)
FPTraceLink get_by_addrs_root(&self, parent: FPTraceLink, addr: &[u64], length: usize)
Vec <u64> get_path_from_addr_rec(&self, link: FPTraceLink, mut vec: Vec<u64>)
root: (FPTraceLink)
FPTraceAddr entries are linked through a FPTraceLink.
A FPTraceAddr can have a parent and multiple children.
FPTraceAddr
addr: u64
visited: bool
children: Vec<FPTraceLink>
parent: FPTraceLink
FPTraceAddr new(addr: u64, children: Vec<FPTraceLink>, parent: FPTraceLink)
u64 get_addr(&self)
bool is_visited(&self)
parent
children

Figure 1: *failure point tree* class diagram after final implementation

The final *failure point tree* implementation shown in Figure 1 works by creating a new tree instance using the `new` method. All possible entries are then added by calling the `add` method with a kernel stack trace and the stack trace's length as the parameters on the instance. The tree proceeds to recursively call the `add_to_parent` method which uses `contains_root` to recursively check if an address is either partially of fully included already. If the address is fully included, `contains_root` will return `None` and `add` will return `false` to signal that the kernel stack trace has not been added as it is already included. Otherwise, `contains_root` will return the last common `FPTraceLink` (this could be the `root` link if there are no common addresses) and `add_root` will proceed to add the branch of remaining stack trace entries as a child of then given `FPTraceAddr`.

Our initial design for the `FPTraceAddr` included a `visited` parameter. In Mumak, the parameter is used for crash image generation. Mumak runs an application, generates a full *failure point tree* and only then iterates over the tree to generate a crash image for a specific leaf and to run the crash analysis for that image. It then marks the leaf as `visited` and proceeds with this iteration until the entire tree is marked as `visited`. Contrary to Mumak, we parse the trace and generate the crash images along the way. The trace entries are deduplicated by trying to insert the stack trace into the *failure point tree* every time a crash image is supposed to be generated.

While the `visited` logic is currently unused in our implementation, a few ideas came up to utilize this in the future, so it was kept in place. This also applies to the `search`, `get_by_addrs` and `get_path_from_addr` methods as well as their internal implementations which were used for testing during development and are currently only in use by the example. A full example of the *failure point tree* in action with all its supported features can be run using `cargo run --example fptree`.

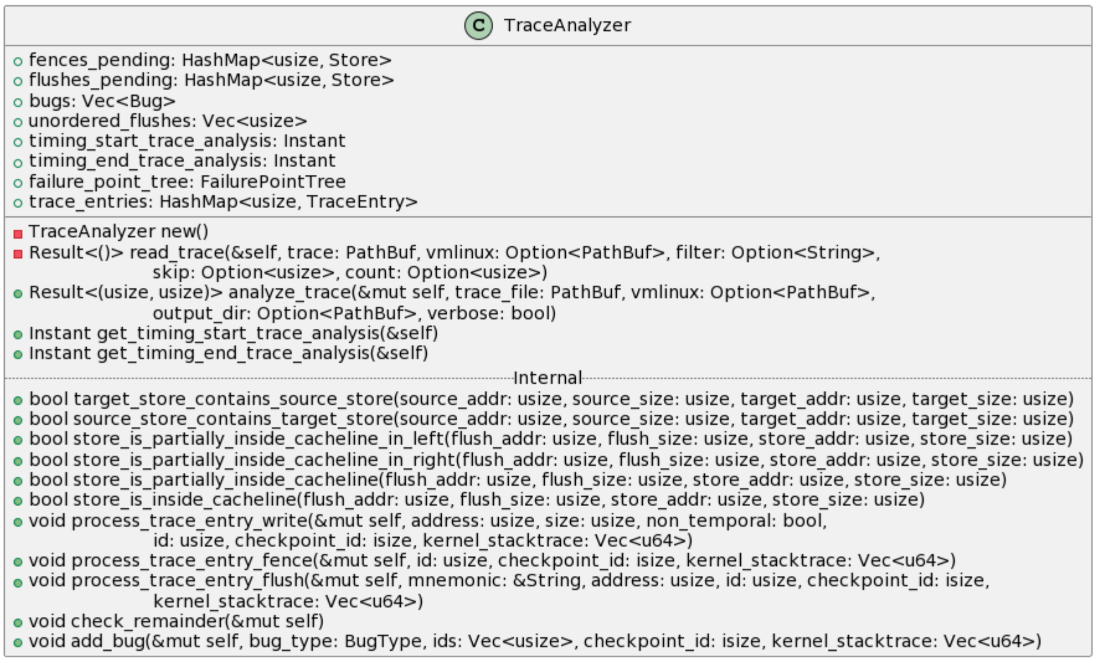
4.2 Trace Analysis

Trace analysis works by pattern matching the order of trace entries affecting addresses within the same range against patterns of known possible bugs.

For example, if an address is flushed, but there were no previous writes to the address, a `RedundantFlush` bug will be reported. All possible bugs are detailed in Section 3.3.

In its first implementation, the trace analyzer was implemented as an optional feature during a `vinter_trace2img` run. If the CLI parameter to run trace analysis is set, the `replay` method will collect a separate vector containing trace entries that will be returned to the `main` method. The `main` method proceeds to create a `TraceAnalyzer` instance and passes the entry vector to its `analyze_trace` method.

After review, we decided to move the tracer to a standalone tool that will read existing trace files. As such, the `TraceAnalyzer` was moved into `vinter_report`. To prepare for that, `vinter_report` was refactored and extended with features in the process. A detailed description on that can be found in Section 4.3.3.2. We merged the new `TraceAnalyzer` with the `TraceAnalyzer` responsible for reading and filtering existing traces that we introduced to `vinter_report` during its rework. We refactored the `analyze_trace` method to take a trace file's path that will be read and parsed, rather than a previously created vector of trace entries.



```
C TraceAnalyzer
○ fences_pending: HashMap<usize, Store>
○ flushes_pending: HashMap<usize, Store>
○ bugs: Vec<Bug>
○ unordered_flushes: Vec<usize>
○ timing_start_trace_analysis: Instant
○ timing_end_trace_analysis: Instant
○ failure_point_tree: FailurePointTree
○ trace_entries: HashMap<usize, TraceEntry>
■ TraceAnalyzer new()
■ Result<()> read_trace(&self, trace: PathBuf, vmlinux: Option<PathBuf>, filter: Option<String>,
          skip: Option<usize>, count: Option<usize>)
● Result<(usize, usize)> analyze_trace(&mut self, trace_file: PathBuf, vmlinux: Option<PathBuf>,
          output_dir: Option<PathBuf>, verbose: bool)
● Instant get_timing_start_trace_analysis(&self)
● Instant get_timing_end_trace_analysis(&self)
Internal
● bool target_store_contains_source_store(source_addr: usize, source_size: usize, target_addr: usize, target_size: usize)
● bool source_store_contains_target_store(source_addr: usize, source_size: usize, target_addr: usize, target_size: usize)
● bool store_is_partially_inside_cacheline_in_left(flush_addr: usize, flush_size: usize, store_addr: usize, store_size: usize)
● bool store_is_partially_inside_cacheline_in_right(flush_addr: usize, flush_size: usize, store_addr: usize, store_size: usize)
● bool store_is_partially_inside_cacheline(flush_addr: usize, flush_size: usize, store_addr: usize, store_size: usize)
● bool store_is_inside_cacheline(flush_addr: usize, flush_size: usize, store_addr: usize, store_size: usize)
● void process_trace_entry_write(&mut self, address: usize, size: usize, non_temporal: bool,
          id: usize, checkpoint_id: isize, kernel_stacktrace: Vec<u64>)
● void process_trace_entry_fence(&mut self, id: usize, checkpoint_id: isize, kernel_stacktrace: Vec<u64>)
● void process_trace_entry_flush(&mut self, mnemonic: &String, address: usize, id: usize, checkpoint_id: isize,
          kernel_stacktrace: Vec<u64>)
● void check_remainder(&mut self)
● void add_bug(&mut self, bug_type: BugType, ids: Vec<usize>, checkpoint_id: isize, kernel_stacktrace: Vec<u64>)
```

Figure 2: `TraceAnalyzer` class diagram after merge

Figure 2 marks existing methods as red squares and our additions as green circles. To process a trace, the `TraceAnalyzer` instance is created, and `analyze_trace` is called with a path to the `trace_file`, an `Option` of a path to the trace's `vmlinux` file, an `Option` to an output folder and a `verbose` boolean. `analyze_trace` proceeds to read the file and will iterate over it until the test begins. This is determined using the `hypercall` trace entries. `Write`, `flush` and `fence` entries within the test range will be added to a `HashMap` that is keyed by their `trace_id` and processed in their individual handling methods.

`process_trace_entry_write` checks if a `write` affects a previous `write` that is still pending a `fence` or `flush`'s storage range and reports an `Overwritten-Unflushed` or `OverwrittenUnfenced` bug if this is the case. It then proceeds to add the `write` to the list of pending `flushes` or, if it is a non-temporal write that is considered flushed already, to the list of pending `fences`. These entries are called `stores` internally, and to prevent confusion we will continue to refer to them by that name.

The logic behind `process_trace_entry_flush` is based on a common CPU's cache line size to be 64 bytes and that a flush will always flush an address' entire cache line. While this may vary for some CPU architectures, Vinter does currently only implement the x86 semantic on which this size is guaranteed. If that changes, the cache line size can be adjusted according. With those criteria, the method will filter the list of pending `flushes` by their address being within the cache line that is to be flushed. That cache line ranges from the `flush`'s address to its address plus 64 bytes. Those `stores` are then differentiated in being either fully or partially contained within the `flush`. If a `store` is fully contained in the `flush`, it is marked as `flushed` and will be removed from the list of pending `flushes` after it was added to the list of pending `fences`. A `store` that is only partially contained in the flush will be marked as `PartiallyFlushed`. `Stores` that are only partially contained in the `flush`'s range for a second time are added to the list of pending `fences` and will be removed from the list of pending `flushes`. If at least one `store` was either fully or partially flushed in this process and the fences mnemonic is either `clflushopt` or `clwb`, it will be added to the list of unordered flushes. In case no `store` was flushed in the process, the flush is considered redundant and will be reported as a `RedundantFlush`.

`process_trace_entry_fence` checks the list of pending fences and reports the `fence` as redundant if there are no pending fences. It will also report an `Unordered-Flushes` bug if the list of unordered flushes contains more than one flush. The list of pending fences and unordered flushes will be cleared afterwards.

Once `analyze_trace` is done processing the trace it will check the lists of pending `flushes` and `fences` and will report their entries as `MissingFlush` or `Missing-Fence` accordingly.

During initial testing it became apparent that many bugs affecting the same code line get reported multiple times as they appear more than once in the trace. As such, we decided to introduce a method to deduplicate the list of bugs using the kernel stack trace and checkpoint ID. While this requires a trace to include a kernel stack trace, a feature to always log a stack trace was introduced in Section 4.3.3.3. For deduplication, the *failure point tree* was extended to store an `Option<Vec<FPTBug>>` inside the `FPTraceAddr`. A `FPTBug` contains the `BugType` as well as the `checkpoint` during which a bug was encountered. This is necessary, as the same kernel stack trace can have multiple bugs of different types during different test phases assigned to it.

While the original Mumak design only contained the basic information of the bug type and the trace entry's ID shown in Figure 3, this was greatly extended upon in our implementation.

```
> vinter_report analyze-trace results/trace.bin
Analyzing Trace...
---
- bug_type: OverwrittenUnflushed
  id: 295752
```

Figure 3: Trace analysis output when following the original Mumak design

Previously we would only know that ID 295752 was overwritten, but not where and why. To change that, the trace analyzer was extended to store vectors of trace IDs instead of counters and single trace IDs to include all trace entries involved in a bug report as shown in Figure 4. In this example, we report an OverwrittenUnflushed bug because the write with ID 295808 writes to the same address as ID 295752 has written to before without the address range being flushed in between.

```
> vinter_report analyze-trace results/trace.bin
Analyzing Trace...
- Bug Type: OverwrittenUnflushed
  Checkpoint: 1
  Responsible Trace Entries:
  - Write { id: 295752 }
  - Write { id: 295808 }
```

Figure 4: Final Trace analysis output with regular output

This is further extended upon using the -v or --verbose parameter. When requesting a verbose bug report, the report contains the full trace entry including the address, size, content, metadata including the kernel stack trace and the information if the write is non_temporal. We can now see in Figure 5 that both entries try to write the same address without being flushed in between. Similar to previous examples, the metadata was shortened for better readability.

```
> vinter_report analyze-trace results/trace.bin -v
Analyzing Trace...
- Bug Type: OverwrittenUnflushed
  Checkpoint: 1
  Responsible Trace Entries:
  - Write { id: 295752, address: 2990248, size: 1, content: [0], non_temporal: false,
    metadata: Metadata { pc: 81129721, in_kernel: true, kernel_stacktrace: [80102328,
    80033853, 80125903, 79687992, 79688156, 79688374, 78849556, 83039589] } }
  - Write { id: 295808, address: 2990248, size: 1, content: [2], non_temporal: false,
    metadata: Metadata { pc: 80088151, in_kernel: true, kernel_stacktrace: [80099160,
    80102328, 80033853, 80125903, 79687992, 79688156, 79688374, 78849556, 83039589] }
    }
```

Figure 5: Final Trace analysis output with verbose output (pc and stack trace shortened)

Figure 6 displays the output after specifying the traces corresponding `vmlinux` image. Similar to `vinter_report`'s existing `read-trace` feature, using the image will display the corresponding kernel symbols to each trace entry's kernel stack trace. This flag can also be used together with the verbose flag.

```
> vinter_report analyze-trace results/trace.bin --vmlinux nova_out/vmlinux
Analyzing Trace...
- Bug Type: OverwrittenUnflushed
  Checkpoint: 1
  Responsible Trace Entries:
  - Write { id: 295752 }
  Kernel Symbols:
      pc: 0xffffffff8122dbf9 ?? at /nova/arch/x86/lib/memset_64.S:66:0
      stack trace:
      #1: 0xffffffff81132eb8 nova_append_dentry at /nova/fs/nova/log.c:1020:8
      #2: 0xffffffff8112233d nova_add_dentry at /nova/fs/nova/dir.c:322:8
      #3: 0xffffffff81138acf nova_mkdir at /nova/fs/nova/namei.c:500:8
      #4: 0xffffffff810cdc38 vfs_mkdir at /nova/fs/namei.c:3819:1
      #5: 0xffffffff810cdcdc done_path_create at /nova/fs/namei.c:3682:2
      #6: 0xffffffff810cddb6 __x64_sys_mkdir at /nova/fs/namei.c:3852:1
      #7: 0xffffffff81001114 do_syscall_64 at /nova/arch/x86/entry/common.c:290:12
      #8: 0xffffffff81400065 ?? at /nova/arch/x86/entry/entry_64.S:184:0
  - Write { id: 295808 }
  Kernel Symbols:
      pc: 0xffffffff8112f757 nova_update_new_dentry at /nova/fs/nova/log.c:331:20
      stack trace:
      #1: 0xffffffff81132258 nova_get_super at /nova/fs/nova/super.h:202:9
      #2: 0xffffffff81132eb8 nova_append_dentry at /nova/fs/nova/log.c:1020:8
      #3: 0xffffffff8112233d nova_add_dentry at /nova/fs/nova/dir.c:322:8
      #4: 0xffffffff81138acf nova_mkdir at /nova/fs/nova/namei.c:500:8
      #5: 0xffffffff810cdc38 vfs_mkdir at /nova/fs/namei.c:3819:1
      #6: 0xffffffff810cdcdc done_path_create at /nova/fs/namei.c:3682:2
      #7: 0xffffffff810cddb6 __x64_sys_mkdir at /nova/fs/namei.c:3852:1
      #8: 0xffffffff81001114 do_syscall_64 at /nova/arch/x86/entry/common.c:290:12
      #9: 0xffffffff81400065 ?? at /nova/arch/x86/entry/entry_64.S:184:0
```

Figure 6: Final Trace analysis output with kernel symbols

4.3 Additional Changes

Apart from the planned changes, we made a few additions during the thesis as they proved to be necessary or useful at a later point. This includes the introduction of new build and test scripts, additional performance logging, code restructures that were required to extend upon them at a later point or small features to make the evaluation process easier.

4.3.1 Documentation and Scripts

As the `PMFS` file system requires `gcc4` to be built, the documentation was updated with a workaround to successfully install the `bc` tool required for the kernel compilation on the deprecated Debian Jessie based `gcc:4` Docker container.

All test scripts were adjusted to use the `results` directory as an output and their documentation was adjusted accordingly.

Furthermore, three new test scripts were introduced.

`run-rust-single-test.sh` has a flag for the crash image generator, the test to run, if a stack trace should be stored and if the output should optionally be in JSON object or extra verbose. It then proceeds to run the single test against the `Nova` and `PMFS` file systems using `vinter_trace2img`.

`run-rust-parallel-all.sh` extends on that by running all available tests, optionally in parallel with a specifiable count of parallel jobs, against `Nova`, `Nova-Protection` and `PMFS` while keeping the same flags as `run-rust-single-test.sh`. The JSON output will return a JSON Object that is alphabetically sorted by test name and is including the total runtimes for each step. It can be converted directly into a table with tools like *json2table* [16].

Adding to that, `run-rust-parallel-ta.sh` can take the result folder of a `run-rust-parallel-all.sh` run and will run trace analysis for all traces in that folder. The script has flags for JSON formatted as well as extra verbose output and allows specifying if kernel symbols should be included in the output or not. Similar to `run-rust-parallel-all.sh`, its JSON output is alphabetically sorted by test name, includes the total runtime for each step and can be converted directly into a table with tools like *json2table* [16].

4.3.2 Analysis

Both, `vinter_trace2img` and `vinter_report` were extended by a `-j`/`--json` CLI flag to provide their statistics' output in a JSON format. This way the results can be parsed by the newly added test scripts to receive a better overview during evaluation. In addition, they were extended with runtime tracking that is shown after a test run.

Both tools were extended with a `-v`/`--verbose` flag to provide a more detailed human-readable output. In case of `vinter_trace2img`, that includes the individual runtimes of each internal step during the test run. For `vinter_report`'s `analyze-trace` feature, this means each entry on the list of `Responsible Trace Entries` contains more information.

4.3.3 Tools

4.3.3.1 report-results.py

During development, we discovered that `report-results.py` may not always include a step's final state in the semantic state list. When analyzing a test run from the default heuristic this would not be apparent because the default heuristic indirectly added the final state to the state list. As this does not always happen when using the *failure point tree* implementation we made sure to always add a checkpoint's final state to the list of different states.

In addition, some internal logic was restructured to properly analyze test results coming from the *failure point tree* implementation. Some image handling required to track dirty lines would previously depend on the `HeuristicApplied` property which is unused with the *failure point tree*. The guard was moved to a later point to allow using the common handling code with the results coming from the *failure point tree* implementation.

4.3.3.2 vinter_report

In preparation to implement the trace analysis, we refactored `vinter_report` to move all actual code into a separate library file. With the plan for extension in mind, `lib.rs` received a `TraceAnalyzer` class that would initially contain the code to parse the trace and has later on been extended by the trace analyzer described in Section 4.2. Moving the code to a seperate class within the lib allows an easy reuse of shared code between the trace reader and the analyzer.

While a trace file includes the test's entire trace including all preparation steps, we are usually only interested in the test's trace part marked by the hypercalls. To allow quickly finding the hypercalls, or any other specific trace entry type for that matter, we added the `--filter <FILTER LIST>` parameter to filter the trace by a comma separated list of trace entry types.

Combined with `read-trace`'s existing `--skip <NUM>` flag to skip a certain number of entries, this allows to limit the trace to the entries of interest. In addition, a `--count <NUM>` parameter to limit the amount of trace entries to consider after the start was introduced.

```
vinter_report read-trace results/vm_nova/test_rename-dir/trace.bin \
--skip 296055 --count 5 --filter write,flush
```

Figure 7: `vinter_report read-trace` example using filter and count parameter

In this example, the output of the command shown in Figure 7 will include all `writes` and `flushes` ranging from the 5 trace IDs between `296055` and `296059`.

4.3.3.3 vinter_trace2img

While `vinter_trace2img` is the easiest way to generate a trace file, our trace analysis requires a kernel stack trace for deduplication. A run with the default heuristic or no heuristic at all would not include a kernel stack trace by default as it is not needed in the regular test and introduces a runtime overhead of up to 91% in the trace step[1]. In order to store a kernel stack trace for a later usage in those cases, we extended `vinter_trace2img` with a CLI flag (`-k`/`--kernel-stacktrace`) to always request and store a stack trace. This flag will be ignored when using the *failure point tree* generator as a kernel stack trace is always stored there by design.

[1]Comparing the average trace step runtime of test_rename on nova-protection when using the default heuristic (2196.4 ms) compared to using the default heuristic and saving the kernel stack trace (4196.2 ms) on the test system. Reference: Appendix A and Section 5.2.1.2.

Chapter 5

Evaluation

In the previous chapter we have described the design and implementation of our additions. We detailed the introduction of a *failure point tree* to use for trace entry deduplication and have described it's implementation into the crash image generator. We have also outlined the implementation of the pattern based trace analyzer.

In this evaluation we will show the efficiency of our work, describe and evaluate possible problems and compare the work to Vinter's original implementation. We will start off by describing the test setup and evaluate the changes done to the crash image generation in regard to our two design goals of runtime improvement and similar bug report results.

The Trace Analyzer will be detailed by outlining its quick runtime and by analyzing some of its reported bugs for sanity. We show how the trace analyzer is a good addition to the *failure point tree* implementation and outline potential issues with the trace analyzer's logic.

5.1 Test Setup

During the Evaluation we will use the existing tests included in Vinter. Their detailed results can be found in Appendix A. For the `vinter_trace2img` results, we ran all of our tests against `Nova`, `Nova-Protection` and `PMFS` and collect the results in a JSON formatted output[1]. These runs were repeated for a total of five consecutive runs and the average in runtime, fence/flush count, crash image count, semantic state count and failed recoveries was calculated.

While the consecutive runs were originally intended to only create an average of the runtime, it became apparent that some tests do not always result in the same set of test-relevant trace entries every run, resulting in different amounts of crash images and bugs during trace analysis.

[1]Using `run-rust-parallel-all.sh`

We calculated the trace analyzer results by running all available tests for `vinter_trace2img` on `Nova`, `Nova-Protection` and `PMFS` using the *failure point tree* generator in parallel[1], followed by running `vinter_report`'s `analyze-trace` feature for each test result. We collected the JSON output containing the bug count, the count of relevant trace entries and the runtime for each individual test[2]. We repeated this procedure for a total of five consecutive times and calculated the average of the output.

The test system uses a modified artifact evaluation VM `vinter.qcow2` [12] and is configured as shown in Table 5.1.

Host CPU	AMD Ryzen Threadripper 3970X (32 Cores @4.1GHz All-Core)
Host RAM	4x 32 GB DDR4-3600 CL16
Host OS	Ubuntu 22.04.3 (6.2.0-35-generic)
Host Drive	Samsung 980 Pro
VM OS	Fedora 34 (Cloud Edition) x86_64 (5.17.6-100.fc34.x86_64)
VM CPU	32 Threads
VM RAM	64 GB
VM Tech	KVM

Table 5.1: Test system specs with VM config

We upgraded the VM's packages to the newest available[3] upgraded Rust to version 1.72.1. We disabled *SELinux* due to complications with `podman` when building the `PMFS` kernel, and adjusted the command to build the `PMFS` kernel as documented in Section 4.3.1.

In comparison to Vinter's documentation, we adjusted the `qemu` command for the host system's OS and increased the RAM as well as the CPU count. Since Ubuntu uses `qemu-system-x86_64` rather than `qemu-kvm`, the accelerator was set to `kvm` as `qemu-system-x86_64`'s default `tcg` is unbearably slow and not designed for general usage. This results in the final command shown in Figure 1.

```
qemu-system-x86_64 -m 64G -smp 32 -display none -accel kvm \
-serial mon:stdio -device e1000,netdev=net0 \
-netdev user,id=net0,hostfwd=tcp::2222-:22 vinter.qcow2
```

Figure 1: Adjusted `qemu` command for the test system

[2]Using `run-rust-parallel-ta.sh`
[3]as of Oct 26 03:08:31 AM UTC 2023

5.2 Crash Image Generation

While we introduced a *failure point tree* to deduplicate crash images, this approach also skips the heuristic to create additional crash images that could likely be of interest. We will continue to evaluate the *failure point tree's* deduplication by itself and the full implementation compared to the default implementation.

5.2.1 Runtime Performance

During the runtime's evaluation we will outline how our changes to the crash image generation affect each test run's step. Our expectation is a significant improvement in overall runtime, especially for long tests. Due to the additional collection of a kernel stack trace in the trace, we account for a small increase in total runtime on smaller tests that do not profit from the *failure point tree's* trace entry deduplication.

5.2.1.1 Standalone Failure Point Tree

To evaluate the *failure point tree's* standalone effect on the runtime, we modified the code to skip the *failure point tree* handling [21]. This results in creating up to two crash images, one without pending writes and, if existent, one with the pending writes included, for each fence or flush that is preceded by a write. This will be referred to as `FPT-SKIP` from now on. We compared `FPT-SKIP` against a regular run using the *failure point tree* to deduplicate trace entries. A regular run using the *failure point tree* will be referred to as `FPT` from now on.

Figure 2 displays a list of selected test runs and the overall runtime average over all tests. Each test has a group of two bars assigned to it. The upper bar visualizes the runtime on `FPT-SKIP` and the lower bar visualizes the runtime on `FPT`. The bar is separated into three segments, to cover the three steps in a Vinter test run. The colors for the trace step, the crash image generation step and the tester step, during which the tester will analyze the crash images to find different semantic states, are described in the legend.

While we will continue to evaluate and explain the results shown in the graph in detail, the graph does provides a quick outlook on what kind of runtime improvement to expect from the *failure point tree*.

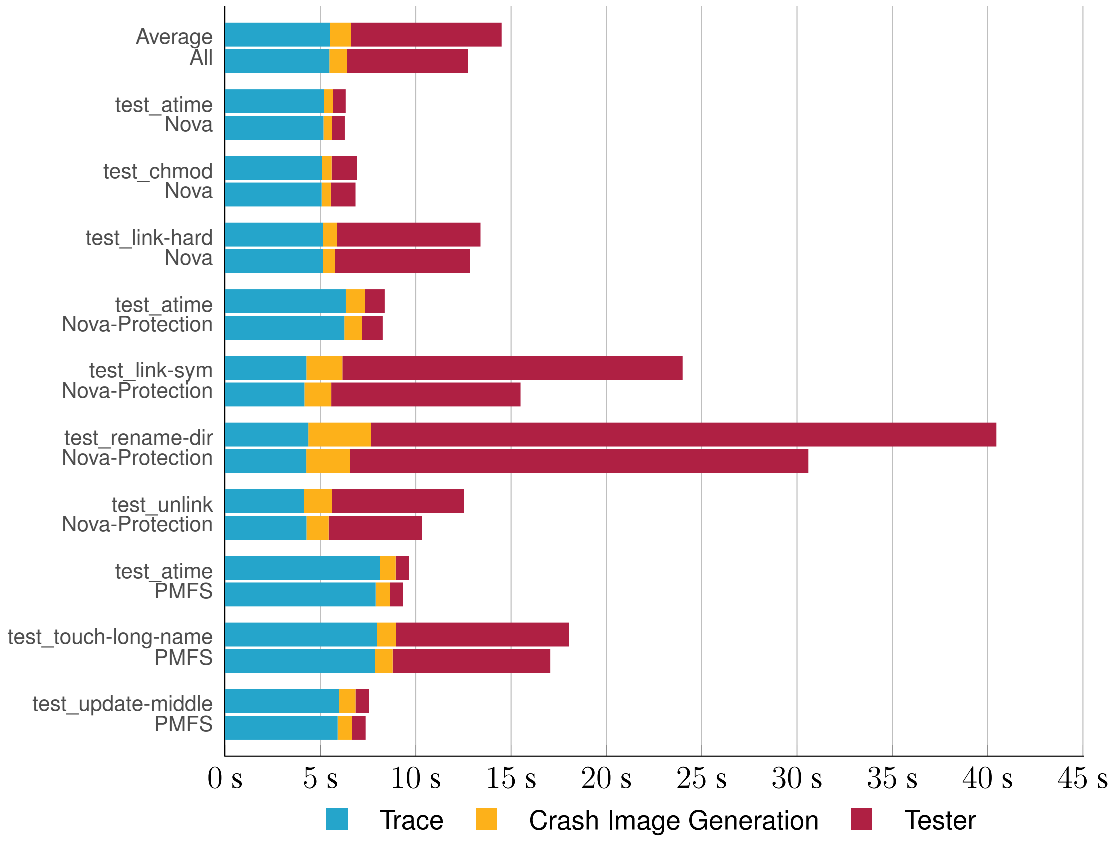


Figure 2: Runtime comparison between `FPT-SKIP` (top) and `FPT` (bottom)

As `FPT-SKIP` and `FPT` only differ in the crash image generation, the trace step remains identical in both tests.

Despite being identical, the trace step runtime in our tests show a range difference between -3% for `test_unlinked` on `Nova-Protection` and +3% for `test_atime` on `PMFS` across all tests. The average discrepancy in trace runtime between `FPT` and `FPT-SKIP` is ±1.1%. The the total trace runtime difference between `FPT` and `FPT-SKIP` is 0.7%.

We use these numbers to set the margin of error in our evaluation to ±5.0%. This margin will help us to determine if a tests runtime change is only a system load related difference in runtime or an actual change.

Analyzing the crash image generation step's runtime shows us that, even when `FPT` and `FPT-SKIP` generate the same amount of crash images, the runtime still profits from the *failure point tree*'s deduplication by between 3.9% for `test_chmod` on `Nova` and 10.6% for `test_update-middle` on `PMFS`. That is an average improvement of 6.8% per test. This happens, because the crash image generator uses a `HashMap` to organize crash images internally. While the *failure point tree* would skip the crash image generation after determining that a kernel stack trace is already contained, `FPT-SKIP` will always create a crash image and insert it into the `HashMap`, overwriting the equal crash image that was contained in the `HashMap` before. This means, that the amount of crash images does not change, but the work to create a crash image twice contributed to the increase in runtime for `FPT-SKIP`.

The potential runtime overhead introduced by the failure point tree creation and insertion is small enough to not be of negative effect in the individual steps runtime. In our comparison, the overhead introduced by the additional crash image generation was higher than the potential overhead from the tree itself for every test.

Considering out margin of error, we can therefore safely say that the *failure point tree* does always improve the runtime and that the improvements of 3.9%-10.6% on the runtime even with the same the amount of resulting crash images can be considered real.

We will proceed to look into the changes with actual changes in the crash image count. The difference, if any, in crash image count for the tests mentioned in our evaluation are shown in Figure 3.

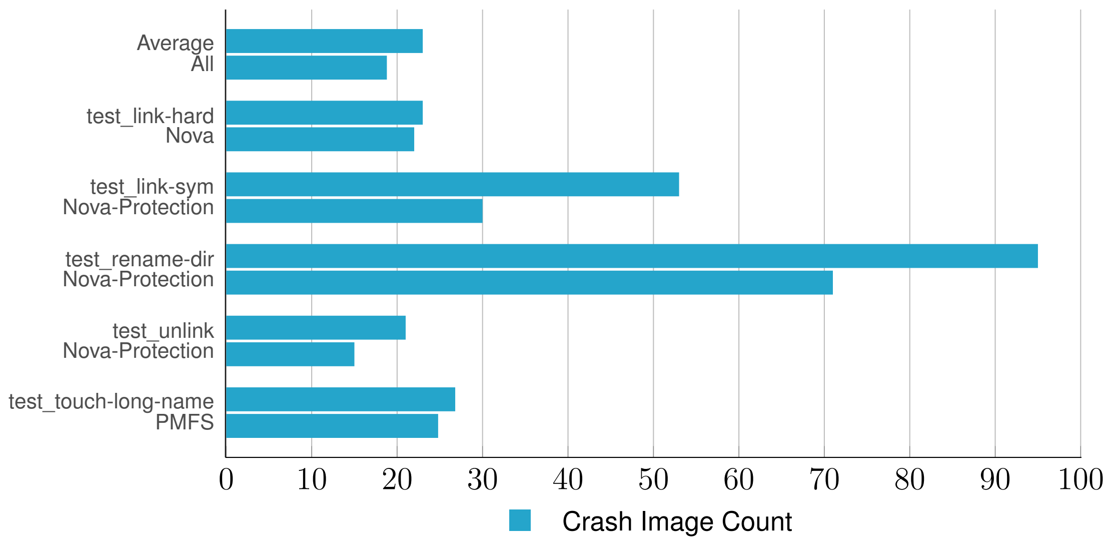


Figure 3: Crash image count comparison between `FPT-SKIP` (top) and `FPT` (bottom)

For tests with an actual change in crash image count, the crash image generation's runtime is decreased by between 6.4% for `test_touch-long-name` on `PMFS` and up to 30.2% for `test_rename-dir` running on `Nova-Protection`. The average change was observed to be 12.6% per test. The crash image generators runtime on all tests combined has been improved by 15.6%, from a total of 52.8 seconds down to 44.5 seconds.

This is, however, not the only step in which the *failure point tree* is responsible for runtime improvements. The tester processes every crash image and checks it for varying semantic states. As this is a single threaded process, a decrease in crash images will directly result in a decrease in the tester step's runtime.

Across all tests, the changes in runtime for the semantic state discovery range from a 3.8% slowdown for `test_atime` to a 44.4% improvement for `test_link-sym`, both on `Nova-Protection`. Considering that `test_atime` only has three crash images on both, `FPT` and `FPT-SKIP`, the 3.8% increase in tester runtime between `FPT` and `FPT-SKIP` can be justified as equal within the margin of error. In total, this steps runtime is improved by 19.7%, from 6.3 minutes down to 5.1 minutes.

When only considering tests that have an actual change in the crash image count, the improvement is between 5.7% for `test_link-hard` on `Nova` with `22` crash images on `FPT` against `23` crash images on `FPT-SKIP` and the aforementioned 44.4% for `test_link-sym` on `Nova-Protection` with `30` crash images on `FPT` against `53` images on `FPT-SKIP`. The average per-test improvement in runtime for tests with a change in crash image count is 18.6%. For reference, that is an absolute improvement from 17.8 s down to 9.9 s in the tester for `test_link-sym`.

These improvements in the crash image generation as well as the tester runtime are reflected to the total runtime as well. The final runtime when using `FPT` compared to `FPT-SKIP` is, at worst, even for `test_atime` running on `Nova`. This test has a rather small runtime of around 6.3 seconds to begin with and has only three crash images on both `FPT` and `FPT-SKIP`. For tests in which the *failure point tree* decreased the amount of crash images that have to be generated and analyzed, the improvement in total runtime can be up to 35.4%, as it is for `test_link-sym` on `Nova-Protection`. While the improvement in crash image generation is only 25.4% for this test, the big decrease in crash images results in the aforementioned improvement of 44.4% in the tester step and an overall improvement in runtime of 35.4%. In absolute numbers, this is an improvement from 24.0 seconds down to 15.5 seconds for this test.

Using just the *failure point tree* rather than generating a crash image at every given moment, the overall runtime for all tests of 11.6 minutes is decreased by 12.2% down to 10.2 minutes.

5.2.1.2 Full Failure Point Tree Implementation

With the *failure point tree* approach, we will only generate up to two crash images per trace entry. One with no pending writes persisted and, if they exist, one with all pending writes persisted.

Vinter's default heuristic, from now on referred to as `Default`, additionally generates crash images that could likely be of interest. In most cases, this will result in an increase in runtime.

Similar to the graph in Section 5.2.1.1, Figure 4 displays a selected list of tests. Each group displays the `Default` runtime on the upper, and the `FPT` runtime on the lower bar. Once again, the runtime is separated by the test runs individual steps. The graph gives us an overview of the increase in trace runtime and the improvements in crash image generation and semantic state discovery in the tester. We will proceed to evaluate and explain this behavior in detail.

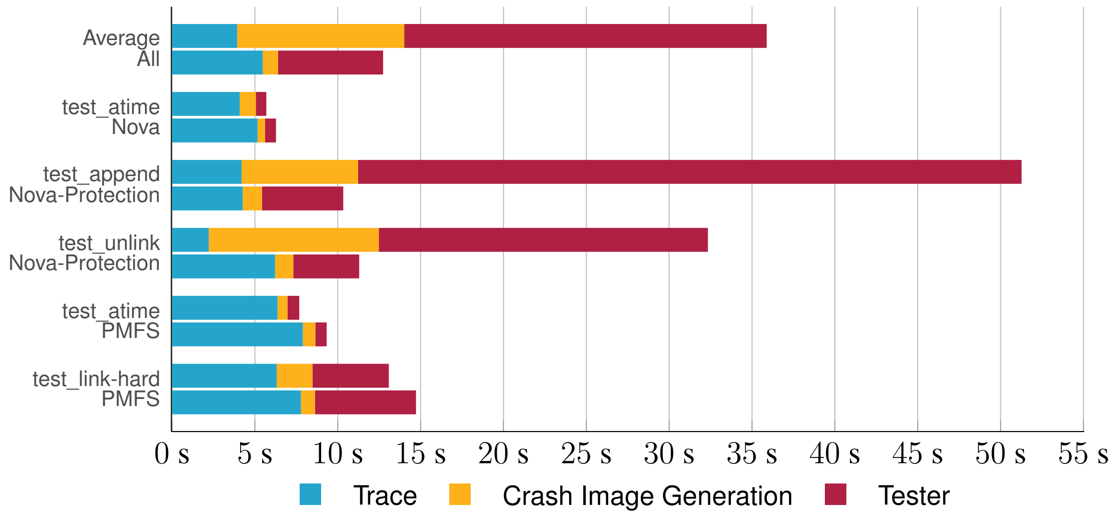


Figure 4: Runtime comparison between `Default` (top) and `FPT` (bottom)

Other than the `Default` implementation, `FPT` requires a kernel stack trace to deduplicate its entries. While collecting a stack trace can also be useful for a later trace analysis, it negatively affects the trace steps runtime. The increase in the trace step's runtime when comparing `FPT` to `Default` ranges from 23.3% for `test_link-hard` on `PMFS` to up to 92.3% for `test_unlink` on `Nova-Protection`. In total for all tests, that is an increase of 44.7% from 3.1 minutes up to 4.4 minutes. The slowdown scales with the amount of trace entries, which means that, the longer the test, the slower the trace step will be.

As crash image count and their improvements between `FPT` and `Default` scale roughly similar to the trace length as well though, the increase in trace time can be compensated and improved in most of the tests.

A comparison of the crash image count for the tests shown in Figure 4 is displayed in Figure 5.

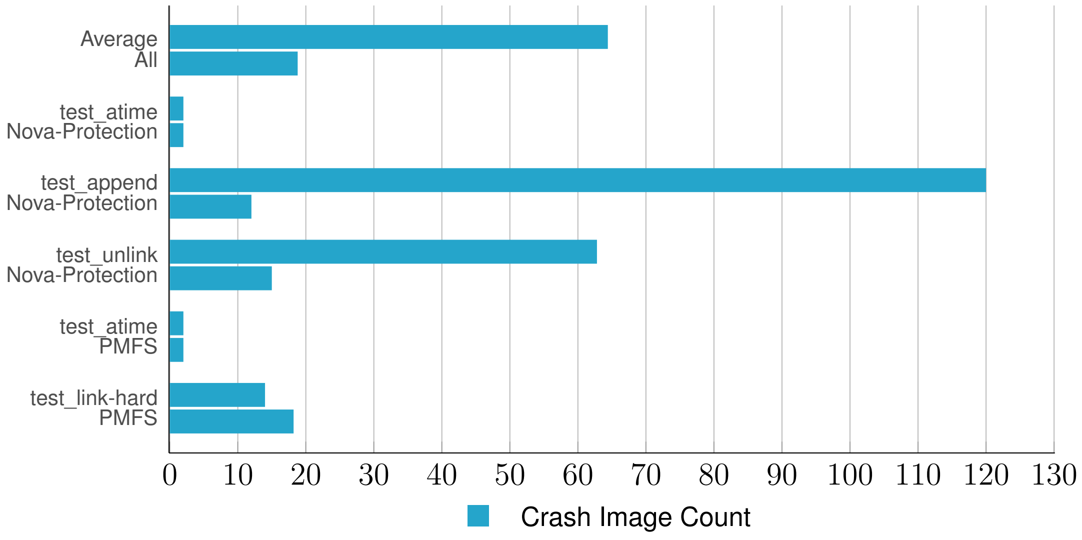


Figure 5: Crash image count comparison between `Default` (top) and `FPT` (bottom)

While `FPT`'s overall runtime change, when compared to `Default`, ranges from a 21.5% slowdown to an 85.7% improvement, the average runtime improvement over all tests is 64.5%. In absolute numbers that is an improvement from 28.7 minutes down to 10.2 minutes.

Out of the 48 tests across the three file systems, only ten are affected by a slowdown when using `FPT` over `Default`. The average slowdown of 11.5% across those tests, however, does only make up a total runtime of an additional 10.1 seconds going from 87.7 up to 97.8 seconds.

Eight out of those ten tests show the same explanation. While crash image generation is significantly faster on `FPT` for all of them, with improvements between 25.1% up to 70.0% and an average improvement of 53.6% per test, `FPT` generated additional crash images. This happens because `FPT` tries to generate crash images for fences and flushes while `Default` only generates crash images after a fence.

As more crash images were generated, the tester had to check more images, and its runtime increased by 7.5% up to 37.2% with a median of 23.0%. Paired with the increase in runtime caused by the kernel stack trace collection in the trace step, the overall runtime is just slower.

The remaining two are `test_atime` on `Nova` and `PMFS`. While both tests had the same amount of fences and flushes as well was crash images as their `Default` equivalent, the `Nova` run showed a 52.8% improvement in crash image generation with an equal runtime for the tester. The 26.1% increase in the tracer step runtime, however, ended up increasing the total runtime by 10.3% for a total of 6.3 seconds on `FPT` up from 5.7 seconds on `Default`.

The test run on `PMFS`, however, is different. It is the only test that shows an increased runtime in the crash image generation step. `test_atime` is a very short test with only a single fence, two crash images and two semantic states. While its increased runtime in the trace is expected, a 25.1% increase in the crash image generation's runtime is not explainable until looking at the comparison of the same test on `Default` and a `Default` run with an included kernel stack trace (`Default+ST`). These tests show the same increase in crash image generation despite not even using a stack trace in their crash image generation. Upon closer inspection, we discover that the crash image generation increase might, in relation, be rather big, but in absolute numbers this is an increase from 603.8 ms to 755.4 milliseconds. At the same time, the trace file's size increases from 6 MB to 7.6 MB just by collecting the kernel stack trace. As the crash image generation step includes reading and parsing the trace file, the increase in runtime can be explained by the additional time it takes to read the file. The remaining increase in total runtime is once again explained by the increase of runtime during the trace step.

For the remaining tests that were not affected by a slowdown, the overall runtime has improved by 5.6% up to 85.7% with an average improvement of 53.7%. In total runtime, this is an 68.5% improvement in runtime from 27.2 minutes down to 8.6 minutes.

While those tests also have the aforementioned trace step slowdown of 23.7% to 92.2% with an average of 49.2% per test, the improvement of 37.6% to 96.5% with a per test average of 86.3% during the crash image generation paired with the 5.6% to 90.1% improvement with an per test average of 58.7% in the tester step make up for the runtime increase in the trace step.

Especially on long tests we can observe big improvements in runtime thanks to an up to 90.0% decrease in crash images. That is a difference of 108 crash images coming down to 12 on `FPT` from the 120 images on `Default` for `test_append` on `Nova-Protection`.

This difference is directly mirrored to the improvements in runtime during the semantic state, which is the same 90.1% in case of the aforementioned `test_append`.

Comparing the decrease in crash images and the testers runtime, no matter if positive or negative, we can, within our accepted margin of error, say, that improvements are directly equal to each other. This is confirmed by the results when comparing `FPT` to `FPT-SKIP`.

To summarize, while some of the smaller tests have a slightly increased runtime with `FPT`, the improvements in runtime, especially in long tests, justify the usage of `FPT` over `Default` when purely having runtime performance in mind.

5.2.2 Result Quality

We will continue to evaluate the *failure point tree* implementation in terms of actual bug detection quality. We begin by evaluating the *failure point tree's* deduplication and then proceed to evaluate the full implementation in the crash image generator. In regard to the result quality, we specifically evalute ommitting Vinter's heuristic responsible for creating additional crash images that likely are of interest to discover additional semantic states in our *failure point tree* implementation.

5.2.2.1 Standalone Failure Point Tree

When using `FPT` in comparison to just inserting the crash images with `FPT-SKIP`, the amount of crash images is between even in a `test_atime` run for all file systems and up to 43.4% less, going from `53` down to `30` images in a `test_link-sym` run for `Nova-Protection`. On average this represents a decrease in crash images by 13.8% per test and a decrease in total crash images create by over 18.5% going from a total of 1105.2 crash images down to 900.8. A visual representation of that decrease is included in Section 5.2.1.1. While even more potential crash images are attempted to be created without the tree on `FPT-SKIP`, Vinter uses a `HashMap` to organize the crash images internally. If a generated crash image is exactly the same as an existing crash image, it will not be handled twice.

Despite that decrease in crash images, we confirmed that `FPT` delivers the same semantic states as `FTP-SKIP`. This means, that the deduplication caused by the *failure point tree* does not affect the bug result quality at all.

5.2.2.2 Full Failure Point Tree Implementation

In addition to the *failure point tree's* deduplication, our implementation will attempt to generate crash images if a trace contains a write followed by a flush or a fence compared to only attempting to generated crash images for a write-fence combination on the default implementation. The *failure point tree* implementation does, however, not use the heuristic to generate crash images that likely are of additional interest to discover additional semantic states. While we have shown that this does greatly improve the runtime, the `FPT` implementation was not able to fully reproduce the same results delivered by the `Default` implementation.

We will evaluate the tests ordered by the potential bugs that were discovered. We categorize the tests by their amount of semantic states and failed recoveries into three categories. In those categories, the tests are ordered by having a single final state (SFS) and their atomicity. The tables use abbreviations for the difference in Semantic States (ΔSem.) and the difference in failed recoveries (ΔFR) and list the tests' single final state (SFS) and atomicity in a `FPT`/`Default` format. Atomic tests that have a single final state are considered to not contain bugs and are therefore of equal result without further confirmation. Tests that are reported to contain bugs are reviewed by their similarities or differences in their single final states and their potential cause in the code as well as their differences between the semantic states that imply atomicity bugs using `report-results.py analyze --verbose --diff` individually.

Equal Results The first category contains tests with equal amount of semantic stats and failed recoveries between `FPT` and `Default`. This includes the tests in which no bugs were discovered by `FPT` and `Default` shown in Table 5.2 and tests with the same bugs reported shown in Table 5.3.

For the tests where no bugs were reported we conducted no further analysis as both implementations report no bugs at all and are therefore of equal result.

Table 5.2: Tests with no discovered bugs on both, `FPT` and `Default`

VM	Test	ΔSem.	ΔFR	SFS	Atomic
vm_nova	test_atime	0	0	✓/✓	✓/✓
vm_nova	test_chmod	0	0	✓/✓	✓/✓
vm_nova	test_chown	0	0	✓/✓	✓/✓
vm_nova	test_ctime-mtime	0	0	✓/✓	✓/✓
vm_nova	test_mkdir-rmdir	0	0	✓/✓	✓/✓
vm_nova	test_unlink	0	0	✓/✓	✓/✓
vm_nova-protection	test_atime	0	0	✓/✓	✓/✓
vm_nova-protection	test_chmod	0	0	✓/✓	✓/✓
vm_nova-protection	test_chown	0	0	✓/✓	✓/✓
vm_pmfs	test_append	0	0	✓/✓	✓/✓
vm_pmfs	test_atime	0	0	✓/✓	✓/✓
vm_pmfs	test_chmod	0	0	✓/✓	✓/✓
vm_pmfs	test_chown	0	0	✓/✓	✓/✓
vm_pmfs	test_link-hard	0	0	✓/✓	✓/✓
vm_pmfs	test_link-sym	0	0	✓/✓	✓/✓
vm_pmfs	test_rename-long-name	0	0	✓/✓	✓/✓
vm_pmfs	test_touch	0	0	✓/✓	✓/✓

Table 5.3: Tests with the same discovered bugs between `FPT` and `Default`

VM	Test	ΔSem.	ΔFR	SFS	Atomic
vm_nova	test_link-hard	0	0	✓/✓	✗/✗
vm_nova	test_rename	0	0	✓/✓	✗/✗
vm_nova	test_rename-dir	0	0	✓/✓	✗/✗
vm_nova	test_touch	0	0	✓/✓	✗/✗
vm_pmfs	test_ctime-mtime	0	0	✓/✓	✗/✗
vm_pmfs	test_hello-world	0	0	✓/✓	✗/✗
vm_pmfs	test_rename-dir	0	0	✓/✓	✗/✗
vm_pmfs	test_touch-long-name	0	0	✓/✓	✗/✗

For tests that reported a bug, we conducted further analysis. As displayed in Table 5.3, the tests match in their single final state and each of their checkpoint's atomicity. Each test reported to have atomicity bugs was verified to have similar difference between its semantic states in each checkpoint for `FPT` and `Default` to make sure `FPT` found the same kind of atomicity bug as the `Default` approach. This was the case for all tests on that list. As such, we consider these results to be equal.

Additional Semantic State, missing Failed Recovery The next category are tests that report additional semantic states on `FPT` but less failed recoveries when compared to `Default`. When Vinter's tester fails to recover a crash image, the image will be reported as a failed recovery and automatically be considered a guaranteed bug as crash consistency always implies a successful recovery from a crash. As Vinter however also considers a semantic state to be a possible bug, using `FPT` over `Default` should yield the same results, although in different form.

Table 5.4: Tests with additional semantic states and missing failed recoveries on `FPT` compared to `Default`

VM	Test	ΔSem.	ΔFR	SFS	Atomic
vm_pmfs	test_mkdir-rmdir	1	-1	✓/✓	✗/✗
vm_pmfs	test_rename	1.2	-1	✓/✓	✗/✗
vm_pmfs	test_unlink	1.2	-1	✓/✓	✗/✗

Table 5.4 lists the aforementioned tests. The 0.2 difference with `test_unlink` and `test_rename` is likely to be caused by the varying trace output of the `PMFS` file system. It is to be assumed that the result will near the value of a single additional semantic state with a bigger sample size. On detailed inspection, all three tests show the same kind of atomicity bug and the same kind of kernel bug (`kernel BUG at /mnt/pmfs/fs/pmfs/balloc.c:70!`) when trying to recover some of the crash images. Their test results will therefore be considered equal as well.

Missing Semantic States The last class of results are tests with fewer discovered semantic states. We verify if these semantic states still cover the same kind of bugs or if results are omitted.

Table 5.5: Tests with fewer semantic states discovered by `FPT` in comparison to `Default`

VM	Test	ΔSem.	ΔFR	SFS	Atomic
vm_nova-protection	test_append	-1	0	✓/✓	✓/✗
vm_nova-protection	test_ctime-mtime	-1	0	✓/✓	✓/✗
vm_nova-protection	test_mkdir-rmdir	-1	0	✓/✓	✓/✗
vm_nova-protection	test_unlink	-1	0	✓/✓	✓/✗
vm_nova-protection	test_update-middle	-1	0	✓/✓	✓/✗
vm_pmfs	test_update-middle	-5	0	✓/✓	✓/✗
vm_nova-protection	test_hello-world	-1	0	✓/✓	✗/✗
vm_nova-protection	test_link-hard	-2	0	✓/✓	✗/✗
vm_nova-protection	test_rename	-1	0	✓/✓	✗/✗
vm_nova-protection	test_rename-dir	-1	0	✓/✓	✗/✗
vm_nova-protection	test_rename-long-name	-1	0	✓/✓	✗/✗
vm_nova-protection	test_touch	-1	0	✓/✓	✗/✗
vm_nova-protection	test_touch-long-name	-1	0	✓/✓	✗/✗
vm_nova	test_append	-3	0	✗/✗	✗/✗
vm_nova	test_hello-world	-2	0	✗/✗	✗/✗
vm_nova	test_link-sym	-2	0	✗/✗	✗/✗
vm_nova	test_rename-long-name	-1	0	✗/✗	✗/✗
vm_nova	test_touch-long-name	-2	0	✗/✗	✗/✗
vm_nova	test_update-middle	-5	0	✗/✗	✗/✗
vm_nova-protection	test_link-sym	-2	0	✗/✗	✗/✗

As shown in Table 5.5, multiple tests report a test to be atomic and have a single final state on `FPT` but report an atomicity bug on `Default`. In most of those cases this is because FPT was not able to find the one semantic state that would show the inconsistency with missing and mismatching content.

While test_update-middle on PMFS differs by 5 crash images, all of them show the same atomicity issue. The test creates a file, and replaces lines in the middle of that file with the string hohoho using the command shown in Figure 6.

```
echo -n hohoho | dd of=/mnt/myfile seek=171 bs=6 conv=notrunc
```

Figure 6: test_update-middle checkpoint 1 to 2

The resulting semantic state images contain different variations in length of the hohoho string, showing that this step is not atomic. FPT fails to detect any of those states and is therefore not able to report a bug.

Not being able to discover the atomicity bugs in those tests a degradation in test quality between FPT and Default and does not fullfill our goal to achieve a similar quality in bug reports.

We will continue to analyze the tests that report the same result for single final state and atomicity for similarities and make sure both of them report the same kind of bug.

While having less semantic states, all tests with only one single final state (✓) and reported atomicity issues show the same kind of differences and effectively report the same issue. As such, we regard their result as equal.

The result for the remaining tests that report both, multiple single final states and an atomicity issue, the results vary slightly. All of the tests reported the same kind of bugs despite having fewer semantic states. However, test_update-middle and test_hello-world on Nova report the atomicity between checkpoint 2 and 3 as a yellow (atomic) instead of a red non atomic. This happens because the FPT is able to only find two semantic states for that checkpoint range. As those two semantic states however end up as two final states, atomicity can already be considered violated.

The way report-results.py reports the atomicity state is by counting semantic states. The tool reports a test's checkpoint range to be atomic if there is only one SFS and only two semantic states for a checkpoint range. If there are more than two semantic states, the result is reported as non atomic. If there are only two semantic states but multiple SFSs, the tool will report the checkpoint to be (atomic) as the test is, by its definition of atomicity, atomic, but the tool automatically invalidates this result using the parentheses as a checkpoint can not be atomic with multiple SFSs. We confirmed for both tests that the additional semantic states for those checkpoints on the Default implementation show no additional differences but instead just additional variations of the differences discovered by the FPT implementation. As such, we consider these tests to be equal as well.

As a conclusion we can say that, *if* a bug is discovered on FPT, its debugging value will be the same as with Default.

5.3 Trace Analyzer

In this section we will outline quality improvements brought to a Vinter test by the trace analyzer and evaluate the changes we did to it compared to its original design in Mumak. We will also give a quick overview over its runtime and the changes we did to improve that runtime.

5.3.1 Runtime

The trace analyzer is a new feature. There will be no relative runtime comparison but absolute numbers instead. While the trace analyzer itself is rather quick, the requirement for a kernel stack trace in the trace file needs to be taken into account. A `FPT` run does already include the kernel stack trace, but a run of the `Default` heuristic is required to collect an additional kernel stack trace (`Default+ST`) for the trace analyzer to work. As this is not required for the crash consistency test run we consider the increase in runtime to be part of the trace analyzer.

This means, that the runtime for a `FPT` based run is between 155.2 ms for `test_ctime-mtime` on `Nova` and 658.6 ms for `test_touch-long-name` on `Nova-Protection`. The average runtime for a test is 276.7 ms. In comparision, the runtime for a `Default+ST` based run is between 1186.2 ms for `test_link-hard` on `Nova` and 2680.8 ms for `test_rename-dir` on `Nova-Protection` with an average runtime of 1840.6 ms.

As expected, the trace analyzers runtime scales with the amount of trace entries that need to be processed.

During implementation, we decided to implement the *failure point tree* to deduplicate the bugs found by the trace analyzer as many of the bugs we observed were duplicate entries for the same code. An overview of the deduplications efficiency can be seen in Figure 7.

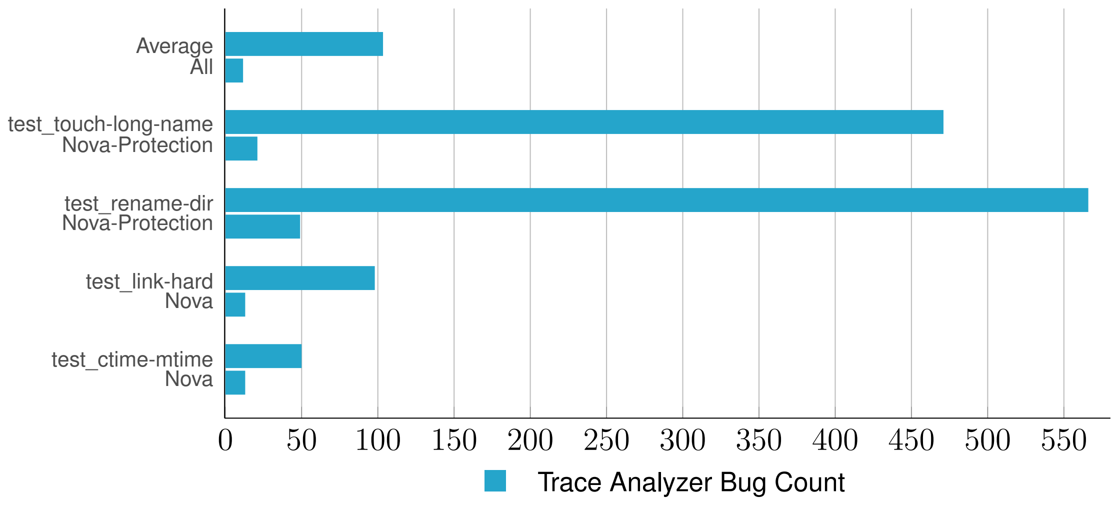


Figure 7: Bug count before (top) and after (bottom) deduplication

The amount of bugs was deduplicated from a range between 0 for `test_atime` on `PMFS` and 566 for `test_rename-dir` on `Nova-Protection`, down to a range of 0 to 49 for the same tests. The average bug count for a test was decreased from 103.4 bugs to 11.6 bugs. This deduplication does not omit any information, as it only deduplicates the same type of bug within the same checkpoint range and with the same kernel stack trace.

While this deduplication introduced the need for a kernel stack trace and therefore the potential addition of additional runtime with `Default+ST`, the runtime of the trace analyzer itself did not increase. In fact, the average runtime even went down to 276.7 ms from 310.0 ms, which appears to be caused by the fewer bugs that need to be written in the report.

5.3.2 Quality

As for its efficiency and result quality, we will take a look at the tests with missing semantic states that reported a bug with the `Default` implementation but not with the `FPT` from Section 5.2.2.2. As a reminder, these are shown in Table 5.6.

Table 5.6: Tests with fewer semantic states discovered by `FPT` in comparison to `Default`

VM	Test	ΔSem.	ΔFR	SFS	Atomic
vm_nova-protection	test_append	-1	0	✓/✓	✓/✗
vm_nova-protection	test_ctime-mtime	-1	0	✓/✓	✓/✗
vm_nova-protection	test_mkdir-rmdir	-1	0	✓/✓	✓/✗
vm_nova-protection	test_unlink	-1	0	✓/✓	✓/✗
vm_nova-protection	test_update-middle	-1	0	✓/✓	✓/✗
vm_pmfs	test_update-middle	-5	0	✓/✓	✓/✗

We start by taking a look at `test_ctime-mtime`, `test_mkdir-rmdir` and `test_unlink` on `Nova-Protection`. These traces report multiple unordered flush bugs within the checkpoint range. Unordered flushes are explicitly listed to be responsible for atomicity bugs. That means that, in this case, the trace analyzer strongly suggests an atomicity bug in the checkpoint range and the trace analyzer is able to potentially detect the atomicity bugs the `FPT` implementation was not able to find when compared to the `Default` implementation.

In case of `test_append` and `test_update-middle` on `Nova-Protection`, the `Default` implementation reported an atomicity bug in the relevant trace range that had not been found by the `FPT`. The trace analyzer reports a missing flush for a write within that checkpoint for both of the tests. The missing flush could explain the lack of atomicity within this range, but it is not a direct proof for the lack of atomicity.

For `test_update-middle` on `PMFS`, the trace analyzer only reports a single redundant fence. This is a performance bug and has no influence on atomicity. In that case, the `FPT` implementation is, even when combined with the trace analyzer, not able to find the same atomicity bug as the `Default` implementation.

This means that the trace analyzer was able to strongly suggest three of the six atomicity bugs missed by the `FPT` implementation, hint at two of the remaining three and could not deliver any additional consistency bug on the last one. Using the ☆ symbol to represent the hints at a potential atomicity bug, our revisioned table is displayed in Table 5.7.

Table 5.7: Tests with fewer semantic states discovered by `FPT` in comparison to `Default` when factoring in the trace analyzer

VM	Test	∆Sem.	∆FR	SFS	Atomic
vm_nova-protection	test_append	-1	0	✓/✓	☆/✗
vm_nova-protection	test_ctime-mtime	-1	0	✓/✓	✗/✗
vm_nova-protection	test_mkdir-rmdir	-1	0	✓/✓	✗/✗
vm_nova-protection	test_unlink	-1	0	✓/✓	✗/✗
vm_nova-protection	test_update-middle	-1	0	✓/✓	☆/✗
vm_pmfs	test_update-middle	-5	0	✓/✓	✓/✗

Chapter 6

Discussion

In the previous chapters we have shown the design and implementation of the *failure point tree* approach into the crash image generator and the pattern based trace analyzer. We proceeded to evaluate the *failure point tree's* runtime improvement and bug detection rate and showed how the trace analyzer can improve that result by finding bugs missed by the *failure point tree* and providing more detail on them. We will now proceed to discuss those results. We will discuss the improvements and drawbacks when using the *failure point tree* implementation over the existing implementation. We will also discuss how using the trace analyzer can make up for some of those drawbacks and potential issues with the trace analyzer itself.

6.1 Failure Point Tree

Extending the crash image generator with the *failure point tree* implementation had two design goals. An improvement in runtime compared Vinter's default implementation and the same results. While it was clear that, by design, the *failure point tree* implementation will not be able to find some of the semantic states discovered through the crash images generated by the heuristic, we were expecting it to show similar results.

The evaluation in Section 5.2.1.2 shows that theses goals were mostly achieved. Thanks to the *failure point tree* implementation's decrease in crash images, the runtime is significantly faster. We were showing an average improvement of 40.1% on an individual test and a decrease in total runtime for all tests of 64.5%. In total runtime, that is a decrease from 28.7 minutes down to 10.2 minutes.

Section 5.2.2.2 shows us that the `FPT` implementation was able to find the same kind of *single final state* bugs as the `Default` implementation. After modifying `report-results.py` to be compatible with the results delivered by the `FPT` in Section 4.3.3.1, we showed that the `FPT` was able to find and report the same code responsible for the dirty lines causing the bug reported. Concluded, `FPT` is able to to find the same consistency bugs as the `Default` implementation. However, `FPT` is not able to find atomicity bugs reported by the `Default` implementation on six of the 48 tests. This means, that the additional crash images generated by the `Default` implementation's heuristic were able to reveal additional semantic states the `FPT` implementation was not able to find.

When combining the *failure point tree* with the trace analyzer, however, we were able to report potential atomicity bugs on three of those six tests and hint at potential issues on two of the remaining three, leaving only one out of 48 tests on which Vinter was not able to find an atomicity bug using the *failure point tree*.

To summarize, using the *failure point tree* implementation will yield big runtime improvements at the cost of a slightly worse bug detection rate.

6.2 Trace Analyzer

The trace analyzer should, by design, be regarded to as a *suggestion*. While its reported *performance bugs* will provide a guaranteed bug in the design, its pattern based trace analysis is not able to check or confirm most of the bugs it reports. For most of the bugs, the trace analyzer can only imply and warn about potential bugs based on the order of trace entries. While those will need to be checked and confirmed manually, the report remains a great help on where to check for potential bugs nonetheless.

However, the trace analyzer leaves some room for improvement. The original design within Mumak contains the ability to detect implicit flushes, but it was commented out [5, 6]. In case of a x86 system, a flush will always flush a full cache line of 64 byte. If there are multiple stores within that line, all of them will be flushed. Every write that was not targeted but affected by that flush however is considered an *implicit flush*. While we assume that it has been commented out because this is how typical `memcpy` routines work, it is undocumented on why this action has been taken in the original Mumak implementation. We discovered that having knowledge of implicit flushes could be of help during the evaluation.

During our tests, we did not discover any unaligned stores that would be affected by a partial flush. However, the way partial flushes are handled requires a closer inspection and potentially a rework. The original design assumes that, if a store is partially flushed twice, it is always stored [4]. This was adapted in our Rust implementation as well [20].

Figure 1 displays the potential problem with that approach. Red displays an unflushed address, green a flushed one. Assuming a store consists of four bytes and the partial flush stores the first two of them. Assuming a second partial flush covers the first two bytes again, the last two bytes are still unflushed. While this store is still clearly partially unflushed, the trace analyzer considers this a successful flush.

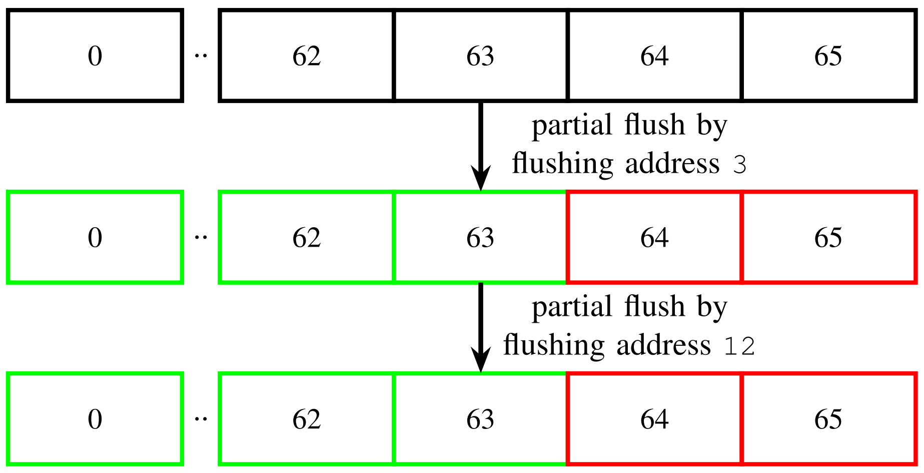

0
..
62
63
64
65
partial flush by flushing address 3
0
..
62
63
64
65
partial flush by flushing address 12
0
..
62
63
64
65

Figure 1: Partial flush handling in `vinter_report`

This design can prevent the trace analyzer from reporting additional unflushed and unfenced writes.

While these improvements could further improve the trace analyzer, it does already provide a helpful addition in file system analysis and represents a valueable addition to Vinter in its current state.

Chapter 7

Conclusion

Non-volatile memory storage is still a comparatively new technology and development for it proves to be difficult. As such, Persistent Memory file systems using that technology require careful testing. Vinter provides a set of tools to do so. In this thesis we extended the crash image generator with Mumak's *failure point tree* approach and added the trace analyzer to Vinter. Both prove to be a valuable addition to Vinter. Using the *failure point tree* to deduplicate the trace entries in order to decrease the amount of crash images that need to be generated and tested provided big improvements in runtime while maintaining almost the same quality in results.

The pattern based trace analyzer offers a new tool to discover an additional set of potential bugs.

We have shown that the trace analyzer can profit from the *failure point tree* as well by using it to deduplicate the reported bugs by their kernel stack trace to prevent reporting the same bug affecting the same code twice.

Further we outlined the performance improvements delivered by our new implementations and show how the trace analyzer can be used to find new bugs or help with those discovered by `vinter_trace2img` already.

7.1 Future Works

Failure Point Tree `visited` logic During implementation it became apparent that we do not require the `visited` handling included by the original *failure point tree* design. In Mumak this logic is used to reiterate over the tree in order to generate crash images and analyze them for each leaf [3]. After a crash image has been generated and analyzed, the leaf will be marked as visited and the iteration will be repeated until all leaves are visited. In our implementation we parse the trace, create crash images along the way and process them after all images were generated. In our implementation, the *failure point tree* is used to check if crash images for a specific kernel stack trace were generated before by checking if a stack trace is already included in the tree during an attempted insertion. If it was not contained before, we add the stack trace to the tree and proceed to generate crash images. As such, marking if a leaf was visited before is not neccesary. However, during the implementation the idea of moving or reusing the tree to an earlier stage in the tool while using the `visited` property alongside the path and leaf search came up. This will need further research that is beyond the scope of this thesis.

Compare Vinter to Chipmunk Chipmunk is a different tool to test a PM file system's crash consistency. Other than Vinter's manually written tests, Chipmunk uses the ACE workload generator [19] and Google's syzcaller grey-box kernel fuzzer [25] to generate test workloads. While it was on our original agenda to compare our *failure point tree* implementation and the default heuristic with the test results delivered by Chipmunk, time contraints did not allow for it. It will be interesting to compare Vinter with both, the *failure point tree* and the default heuristic to Chipmunk to further evaluate its quality in bug discovery.

Trace Analyzer: Implicit Flush As previously mentioned, Mumak's original design for the trace analyzer includes logic to report implicit flushes that is commented out [5, 6]. A careful evaluation and possible rewrite of that code in order to be able to report implicit flushes could be of additional help in bug discovery and analysis.

Trace Analyzer: Rework Partial Flush As mentioned in Section 6.2, the logic to handle partial flushes seems flawed. A careful reevaluation and a potential tracking of which part of a store is partially flushed could further improve the trace analysis results.

Glossary

AppDirect mode Optane DC PMM operation mode to allow persistent storage for software/file systems. 5

memory mode Optane DC PMM operation mode working as a systems main memory and DRAM alternative. 5

NVM non-volatile memory. v, 3, 5–8, 53

Optane DC PMM Intel® Optane™ DC Persistent Memory Module. 3, 5, 55

PM Persistent Memory. v, 3–6, 8–11, 53, 54

SFS single final state. 7, 12, 40, 42–44

List of Figures

1 Potential crash images recovering to one of two semantic states after an atomic operation [11] 7
2 Sample program and corresponding failure point tree [3] 9

1 original Vinter flowchart 12
2 Vinter flowchart with added trace analysis 13
3 *Failure point tree* resulting from trace snippet shown in Table 3.1 15

1 *failure point tree* class diagram after final implementation 22
2 `TraceAnalyzer` class diagram after merge 24
3 Trace analysis output when following the original Mumak design 25
4 Final Trace analysis output with regular output 26
5 Final Trace analysis output with verbose output (pc and stack trace shortened) 26
6 Final Trace analysis output with kernel symbols 27
7 `vinter_report read-trace` example using filter and count parameter 30

1 Adjusted `qemu` command for the test system 32
2 Runtime comparison between `FPT-SKIP` (top) and `FPT` (bottom) 34
3 Crash image count comparison between `FPT-SKIP` (top) and `FPT` (bottom) 35
4 Runtime comparison between `Default` (top) and `FPT` (bottom) 37
5 Crash image count comparison between `Default` (top) and `FPT` (bottom) 38
6 `test_update-middle` checkpoint 1 to 2 44
7 Bug count before (top) and after (bottom) deduplication 46

1 Partial flush handling in `vinter_report` 51

Appendix A

Test Performance Summaries

A detailed documentation on how these results were created can be found in Section 5.1.

Table A.1: Legend

VM	Info about Test VM, named after the file system used
Test	Test scenario
Tech	Info about Crash Image Generator, stack trace etc.
Fences	Amount of Fences (and Flushes, if FPT) during Crash Image generation
CIs	Amount of generated Crash Images
SSs	Amount of different Semantic States found
FR	Failed recoveries during Semantic State Check
Trace (ms)	Duration to run and collect the test trace (in ms)
CI (ms)	Duration to generate Crash Images (in ms)
SS (ms)	Duration to check the Crash Images for different Semantic States (in ms)
Total (ms)	Total Runtime (in ms)
TA Bugs	Amount of bugs found during trace analysis
TA Entries	Amount of trace entries that could potentially cause a bug

A.1 Crash Image Analysis

Table A.2: System 1 Test Average

VM	Test	Tech	Fences	CIs	SSs	FR	Trace (ms)	CI (ms)	SS (ms)	Total (ms)
vm_nova	test_append	Default	5	23	6	0	4017,6	3680,4	7285,8	14985
vm_nova	test_atime	Default	1	2	2	0	4086,6	968,2	635	5690,8
vm_nova	test_chmod	Default	2	3	2	0	4020,8	1568,2	948,6	6538,6
vm_nova	test_chown	Default	2	3	2	0	4041,8	1591	954	6587,6
vm_nova	test_ctime-mtime	Default	14	33	2	0	4056,2	8136,8	10825,4	23019,2
vm_nova	test_hello-world	Default	12	75	7	0	2019,6	9450,8	23707,2	35178,4
vm_nova	test_link-hard	Default	19	36	4	0	4071,6	11093	11559,2	26724,6
vm_nova	test_link-sym	Default	12	102,4	5	0	2012,4	12201	32820,6	47035
vm_nova	test_mkdir-rmdir	Default	22	115,6	3	0	3149,4	15426,2	37640,8	56217,4
vm_nova	test_rename	Default	18	37	4	0	2034	11830,8	11761	25626,4
vm_nova	test_rename-dir	Default	34	175,4	7	0	2129	25379,2	57845,8	85355,2
vm_nova	test_rename-long-name	Default	10	30	5	0	2047,2	6753,8	9662,6	18464,6
vm_nova	test_touch	Default	10	55,6	4	0	4035,8	7859,2	17665,8	29561,6
vm_nova	test_touch-long-name	Default	14	101,4	8	0	4066,6	11460,8	31975,8	47504,2
vm_nova	test_unlink	Default	12	22	2	0	2003	7381,2	6994	16379,2
vm_nova	test_update-middle	Default	5	22	8	0	2070	5158,6	17444,2	24673,8
vm_nova-protection	test_append	Default	6	120	3	0	4195,2	7036,4	40035	51267,8
vm_nova-protection	test_atime	Default	2	7	2	0	4301,4	2288,2	2306	8896,4
vm_nova-protection	test_chmod	Default	5	11,4	2	0	4193,6	4240	3747,6	12181,8
vm_nova-protection	test_chown	Default	5	11,4	2	0	4241,2	4227,2	3764,2	12233,2
vm_nova-protection	test_ctime-mtime	Default	22	67,8	3	0	4296,8	12232,2	22087,8	38618,2
vm_nova-protection	test_hello-world	Default	27	201,8	5	0	2257,8	29471	66368,6	98098,4
vm_nova-protection	test_link-hard	Default	26	95,8	6	0	4269,8	15014,4	30901	50186

Continued on next page

Table A.2 – continued from previous page

VM	Test	Tech	Fences	CIs	SSs	FR	Trace (ms)	CI (ms)	SS (ms)	Total (ms)
vm_nova-protection	test_link-sym	Default	24	223	5	0	2204,8	32143	74043,2	108392
vm_nova-protection	test_mkdir-rmdir	Default	39	194,2	4	0	3355,4	26540	64123,2	94020
vm_nova-protection	test_rename	Default	24	115,2	5	0	2196,4	18654,4	36702,4	57554,2
vm_nova-protection	test_rename-dir	Default	68	317,6	8	0	2290,2	54633,8	107216,4	164141,2
vm_nova-protection	test_rename-long-name	Default	14	54,2	4	0	2209,4	10953,4	17389,6	30553,4
vm_nova-protection	test_touch	Default	22	159,8	5	0	4229,2	25660,2	52788,6	82679
vm_nova-protection	test_touch-long-name	Default	32	169,6	6	0	4270,8	31782,2	56602,6	92656,6
vm_nova-protection	test_unlink	Default	17	62,8	3	0	2219	10258,4	19858,8	32337,4
vm_nova-protection	test_update-middle	Default	6	109	3	0	2287,2	8076,2	60327	70691,6
vm_pmfs	test_append	Default	2,2	3,2	2	0	6220,2	1206,2	1053,6	8480,8
vm_pmfs	test_atime	Default	1	2	2	0	6365,4	603,8	709	7679,4
vm_pmfs	test_chmod	Default	3	5	2	0	6247	1115,2	1649,4	9012,6
vm_pmfs	test_chown	Default	3	5	2	0	6272,6	1116,6	1657,8	9048,2
vm_pmfs	test_ctime-mtime	Default	11,2	18,4	3	0	6365,2	2814,2	6286,6	15467,2
vm_pmfs	test_hello-world	Default	11,6	26,8	3	0	4298,4	4497,8	8801,4	17598,8
vm_pmfs	test_link-hard	Default	10	14	3	0	6310	2169,2	4603	13083,4
vm_pmfs	test_link-sym	Default	6	16	2	0	4248,2	2804	5299,8	12353
vm_pmfs	test_mkdir-rmdir	Default	17	61	8	1	5410,6	5565,8	20560,8	31538
vm_pmfs	test_rename	Default	13,8	41,8	6,8	1	4283,6	5199,2	13606,8	23090,8
vm_pmfs	test_rename-dir	Default	24,8	61,6	5	0	4344,6	9036,4	21193,8	34575,6
vm_pmfs	test_rename-long-name	Default	7	10,2	2	0	4235,6	2236	3499,4	9971,8
vm_pmfs	test_touch	Default	9	20	3	0	6274,4	3198,2	6519	15992,2
vm_pmfs	test_touch-long-name	Default	14,4	28	4	0	6317,4	5393,8	9276,4	20988,8
vm_pmfs	test_unlink	Default	11	15,2	6,8	1	4208,2	2621,6	4810,6	11641,4
vm_pmfs	test_update-middle	Default	2	7	7	0	4382	1220	2365,4	7968
vm_nova	test_append	Default+ST	5	23	6	0	5075,8	3765,2	7378	16219,4
vm_nova	test_atime	Default+ST	1	2	2	0	5159,6	1098,2	639,8	6898,6

Continued on next page

Table A.2 – continued from previous page

VM	Test	Tech	Fences	CIs	SSs	FR	Trace (ms)	CI (ms)	SS (ms)	Total (ms)
vm_nova	test_chmod	Default+ST	2	3	2	0	5110,8	1728,4	955,4	7795,6
vm_nova	test_chown	Default+ST	2	3	2	0	5111	1721,6	954	7787,8
vm_nova	test_ctime-mtime	Default+ST	14	33	2	0	5131,6	8249,6	10827,4	24209,2
vm_nova	test_hello-world	Default+ST	12	75	7	0	3067	9369	23870,6	36307,6
vm_nova	test_link-hard	Default+ST	19	36	4	0	5101,2	11298,6	11675,6	28076,6
vm_nova	test_link-sym	Default+ST	12	86,8	5	0	3037	11161,8	28184,2	42383,6
vm_nova	test_mkdir-rmdir	Default+ST	22	116,6	3	0	4254,4	15171,4	38264,2	57690,8
vm_nova	test_rename	Default+ST	18	37	4	0	3116,8	12045,2	11862,4	27025,6
vm_nova	test_rename-dir	Default+ST	34	174,2	7	0	3163,4	25698,2	57584,8	86447,4
vm_nova	test_rename-long-name	Default+ST	10	30	5	0	3120,8	6912	9643,6	19677,4
vm_nova	test_touch	Default+ST	10	63,8	4	0	5093,2	8002,4	20375,8	33472,2
vm_nova	test_touch-long-name	Default+ST	14	100,6	8	0	5108,6	12632,8	31605,4	49347,6
vm_nova	test_unlink	Default+ST	12	22	2	0	3049,8	7569,2	7042	17662
vm_nova	test_update-middle	Default+ST	5	22	8	0	3126	5310,4	17538,6	25976,4
vm_nova-protection	test_append	Default+ST	6	122,8	3	0	6131,4	7340,8	41165	54638
vm_nova-protection	test_atime	Default+ST	2	6,8	2	0	6303,8	2564	2257,8	11126,8
vm_nova-protection	test_chmod	Default+ST	5	11,6	2	0	6244,4	4376,6	3821,6	14443,6
vm_nova-protection	test_chown	Default+ST	5	11,4	2	0	6206,6	4477,4	3765	14450
vm_nova-protection	test_ctime-mtime	Default+ST	22	67,2	3	0	6239,2	12638,4	22150,4	41029,2
vm_nova-protection	test_hello-world	Default+ST	27	196,6	5	0	4252,8	29757	65075	99085,6
vm_nova-protection	test_link-hard	Default+ST	26	94,2	6	0	6202,4	15445	30712,6	52361,2
vm_nova-protection	test_link-sym	Default+ST	24	215,2	5	0	4135,4	30786	72341	107263,8
vm_nova-protection	test_mkdir-rmdir	Default+ST	39	188,8	4	0	5281,6	28131,8	62651	96065,4
vm_nova-protection	test_rename	Default+ST	24	115,4	5	0	4196,2	19020,2	37072,4	60289,6
vm_nova-protection	test_rename-dir	Default+ST	68	326,6	8	0	4318,2	56274,6	111188,8	171782,8
vm_nova-protection	test_rename-long-name	Default+ST	14	56,2	4	0	4258,8	11223,8	18110	33593,6
vm_nova-protection	test_touch	Default+ST	22	154	5	0	6390,2	25289	51174,8	82854,8

Continued on next page

Table A.2 – continued from previous page

VM	Test	Tech	Fences	CIs	SSs	FR	Trace (ms)	CI (ms)	SS (ms)	Total (ms)
vm_nova-protection	test_touch-long-name	Default+ST	32	166	6	0	6201	33560,8	55849,2	95612,2
vm_nova-protection	test_unlink	Default+ST	17	62,8	3	0	4174,6	10463,6	20037	34676
vm_nova-protection	test_update-middle	Default+ST	6	110,4	3	0	4309,2	8491	64420,8	77222,2
vm_pmfs	test_append	Default+ST	2	3	2	0	7959,4	1303	999,2	10263
vm_pmfs	test_atime	Default+ST	1	2	2	0	7934,6	776	693,6	9405,2
vm_pmfs	test_chmod	Default+ST	3	5	2	0	7861	1292,2	1676,4	10830,6
vm_pmfs	test_chown	Default+ST	3	5	2	0	7861,4	1300	1663,2	10825,6
vm_pmfs	test_ctime-mtime	Default+ST	11	21,6	3	0	7995,4	3028	7459	18483,2
vm_pmfs	test_hello-world	Default+ST	12,2	26,8	3	0	5891,6	5227	8811,6	19931
vm_pmfs	test_link-hard	Default+ST	10	14	3	0	7865,6	2364,2	4665,4	14896,2
vm_pmfs	test_link-sym	Default+ST	6	16	2	0	5974,4	3030,8	5365,4	14371,6
vm_pmfs	test_mkdir-rmdir	Default+ST	17	61	8	1	7060,4	5768,6	20776,8	33606,4
vm_pmfs	test_rename	Default+ST	13,8	34	7	1	5931	5560	11239,6	22731,6
vm_pmfs	test_rename-dir	Default+ST	26,2	59	5	0	6042	9493,4	20482,4	36018,8
vm_pmfs	test_rename-long-name	Default+ST	7	9	2	0	5922,4	2443	3089	11455,4
vm_pmfs	test_touch	Default+ST	9	21,2	3	0	7991,2	3421,2	6989,8	18403,2
vm_pmfs	test_touch-long-name	Default+ST	14,2	28	4	0	7932,6	5207,4	9401	22542,4
vm_pmfs	test_unlink	Default+ST	11,4	16	7,4	1,2	5902,2	3139,4	5084,4	14127
vm_pmfs	test_update-middle	Default+ST	2	7	7	0	5934,6	1401	2382,4	9718,8
vm_nova	test_append	FPT	5	9	3	0	5090,2	508,2	2886	8485,6
vm_nova	test_atime	FPT	1	2	2	0	5157,8	457	658,8	6274,2
vm_nova	test_chmod	FPT	3	4	2	0	5058,8	479,4	1301,2	6840,6
vm_nova	test_chown	FPT	3	4	2	0	5076,6	477,4	1301,8	6857,2
vm_nova	test_ctime-mtime	FPT	17	15	2	0	5150,8	580,2	4947,2	10679,2
vm_nova	test_hello-world	FPT	17	19	5	0	3082,6	632,2	6035,6	9750,8
vm_nova	test_link-hard	FPT	23	22	4	0	5126,8	634,2	7092,2	12854,2
vm_nova	test_link-sym	FPT	16	19	3	0	3044,4	604,2	6091,8	9742

Continued on next page

Table A.2 – continued from previous page

VM	Test	Tech	Fences	CIs	SSs	FR	Trace (ms)	CI (ms)	SS (ms)	Total (ms)
vm_nova	test_mkdir-rmdir	FPT	29	27	3	0	4197,2	700,8	8745,8	13644,6
vm_nova	test_rename	FPT	24	23	4	0	3117,8	655,4	7298	11072,6
vm_nova	test_rename-dir	FPT	48	46	7	0	3180,6	893,6	15071,2	19146,6
vm_nova	test_rename-long-name	FPT	14	18	4	0	3096,2	586,8	5798,8	9482,8
vm_nova	test_touch	FPT	15	15	4	0	5079	578,6	4744,4	10403,4
vm_nova	test_touch-long-name	FPT	20	25	6	0	5029,4	669,4	7811,8	13512
vm_nova	test_unlink	FPT	14	13	2	0	3070,4	554,8	4144,4	7770,6
vm_nova	test_update-middle	FPT	5	8	3	0	3118,4	514,8	5565	9198,8
vm_nova-protection	test_append	FPT	8	12	2	0	6216,6	1102,8	3974,8	11295,2
vm_nova-protection	test_atime	FPT	2	3	2	0	6254,2	937,4	1073,2	8266
vm_nova-protection	test_chmod	FPT	7	6	2	0	6239	1005,2	2006,8	9251,8
vm_nova-protection	test_chown	FPT	7	6	2	0	6220,6	1007	2041	9269,6
vm_nova-protection	test_ctime-mtime	FPT	31	21	2	0	6304,6	1300	7065,6	14670,8
vm_nova-protection	test_hello-world	FPT	36	33	4	0	4227,2	1474	10838,8	16541,2
vm_nova-protection	test_link-hard	FPT	35	23	4	0	6244,8	1333,2	7572,6	15152
vm_nova-protection	test_link-sym	FPT	32	30	3	0	4159,2	1410	9926,8	15497
vm_nova-protection	test_mkdir-rmdir	FPT	53	39	3	0	5278,6	1640,6	13077	19997
vm_nova-protection	test_rename	FPT	37	28	4	0	4163,8	1440,8	9172,8	14778,6
vm_nova-protection	test_rename-dir	FPT	95	71	7	0	4258,2	2293	24045,6	30598
vm_nova-protection	test_rename-long-name	FPT	22	18	3	0	4217,4	1234,8	6009,4	11462,6
vm_nova-protection	test_touch	FPT	31	25	4	0	6183	1362	8217,6	15763,6
vm_nova-protection	test_touch-long-name	FPT	43	32	5	0	6259,4	1527,8	10613,4	18402
vm_nova-protection	test_unlink	FPT	22	15	2	0	4266,2	1161,8	4904,2	10333,2
vm_nova-protection	test_update-middle	FPT	8	11	2	0	4378,8	1126	8151	13656,8
vm_pmfs	test_append	FPT	2,4	3,4	2	0	7826,6	769,2	1132,8	9729,6
vm_pmfs	test_atime	FPT	1	2	2	0	7893	755,4	681,6	9331,2
vm_pmfs	test_chmod	FPT	5	6	2	0	7893,2	765,2	1984,2	10643,6

Continued on next page

Table A.2 – continued from previous page

VM	Test	Tech	Fences	CIs	SSs	FR	Trace (ms)	CI (ms)	SS (ms)	Total (ms)
vm_pmfs	test_chown	FPT	5	6	2	0	7787,6	771	2007,2	10566,8
vm_pmfs	test_ctime-mtime	FPT	13,8	16	3	0	7882,2	848,4	5480,8	14211,8
vm_pmfs	test_hello-world	FPT	16	20	3	0	5809,8	894,6	6521,8	13227,4
vm_pmfs	test_link-hard	FPT	13,2	18,2	3	0	7780,2	844,8	6098,6	14724,4
vm_pmfs	test_link-sym	FPT	10	15	2	0	5802	858,2	5001,6	11662,6
vm_pmfs	test_mkdir-rmdir	FPT	23	31	9	0	7061,8	954,4	10250,2	18267
vm_pmfs	test_rename	FPT	17	24	8	0	5888,8	899	7685	14473,6
vm_pmfs	test_rename-dir	FPT	35	44,4	5	0	5971,6	1072	15272	22317
vm_pmfs	test_rename-long-name	FPT	9	12	2	0	5787	840,2	4077,4	10705,2
vm_pmfs	test_touch	FPT	14	17	3	0	7971,8	852,6	5548,8	14374,4
vm_pmfs	test_touch-long-name	FPT	20,6	24,8	4	0	7863	922,2	8273,8	17059,6
vm_pmfs	test_unlink	FPT	14	17	8	0	5863	856,6	5426,8	12147,4
vm_pmfs	test_update-middle	FPT	2	2	2	0	5899,4	761,6	707,6	7369,2
vm_nova	test_append	FPT-SKIP	5	11	3	0	5073	546,4	3603,4	9223,8
vm_nova	test_atime	FPT-SKIP	1	2	2	0	5174,6	480,4	667,2	6323,8
vm_nova	test_chmod	FPT-SKIP	3	4	2	0	5084,8	499	1333,4	6918,4
vm_nova	test_chown	FPT-SKIP	3	4	2	0	5154,2	500,6	1312,4	6968,2
vm_nova	test_ctime-mtime	FPT-SKIP	17	16	2	0	5130,8	662,8	5302,2	11096,4
vm_nova	test_hello-world	FPT-SKIP	17	22	5	0	3036,6	708,8	7073,8	10820,2
vm_nova	test_link-hard	FPT-SKIP	23	23	4	0	5131,4	737,6	7527	13396,8
vm_nova	test_link-sym	FPT-SKIP	16	27	3	0	3035,2	700,8	8774	12511,2
vm_nova	test_mkdir-rmdir	FPT-SKIP	29	29	3	0	4213,4	816,8	9593,2	14624,8
vm_nova	test_rename	FPT-SKIP	24	26	4	0	3110,2	752,4	8366	12229,6
vm_nova	test_rename-dir	FPT-SKIP	48	48	7	0	3173,6	1061	16109,6	20345,4
vm_nova	test_rename-long-name	FPT-SKIP	14	25	4	0	3124	672,2	8179,6	11976,8
vm_nova	test_touch	FPT-SKIP	15	16	4	0	5092,8	643,6	5174,6	10912,2
vm_nova	test_touch-long-name	FPT-SKIP	20	32	6	0	5161	765,6	10216	16143,6

Continued on next page

Table A.2 – continued from previous page

VM	Test	Tech	Fences	CIs	SSs	FR	Trace (ms)	CI (ms)	SS (ms)	Total (ms)
vm_nova	test_unlink	FPT-SKIP	14	14	2	0	3082,8	633	4547,4	8264,8
vm_nova	test_update-middle	FPT-SKIP	5	10	3	0	3166,8	558,2	6256,4	9982
vm_nova-protection	test_append	FPT-SKIP	8	14	2	0	6187,4	1187,2	4675,8	12051,4
vm_nova-protection	test_atime	FPT-SKIP	2	3	2	0	6326,4	1009,6	1033,8	8371
vm_nova-protection	test_chmod	FPT-SKIP	7	8	2	0	6249	1108,8	2729,2	10088,2
vm_nova-protection	test_chown	FPT-SKIP	7	8	2	0	6285	1112,6	2688,4	10086,8
vm_nova-protection	test_ctime-mtime	FPT-SKIP	31	28	2	0	6260,4	1684,2	9707,8	17653,4
vm_nova-protection	test_hello-world	FPT-SKIP	36	50	4	0	4189,2	1974,8	16826,8	22991,8
vm_nova-protection	test_link-hard	FPT-SKIP	35	30	4	0	6305	1767,8	10079,4	18153,2
vm_nova-protection	test_link-sym	FPT-SKIP	32	53	3	0	4262,6	1889,4	17847	24000
vm_nova-protection	test_mkdir-rmdir	FPT-SKIP	53	53	3	0	5353,2	2246,6	18111,2	25712
vm_nova-protection	test_rename	FPT-SKIP	37	42	4	0	4202,6	1890,2	14018,2	20111,8
vm_nova-protection	test_rename-dir	FPT-SKIP	95	95	7	0	4364,8	3286,6	32808	40460,2
vm_nova-protection	test_rename-long-name	FPT-SKIP	22	25	3	0	4171,8	1501,6	8471,6	14145,8
vm_nova-protection	test_touch	FPT-SKIP	31	33	4	0	6344	1778	11002,2	19125
vm_nova-protection	test_touch-long-name	FPT-SKIP	43	44	5	0	6221,4	2013,4	14870,4	23106,2
vm_nova-protection	test_unlink	FPT-SKIP	22	21	2	0	4133,6	1476,4	6918	12529,2
vm_nova-protection	test_update-middle	FPT-SKIP	8	13	2	0	4326,6	1213,4	8945	14485,6
vm_pmfs	test_append	FPT-SKIP	2,2	3,2	2	0	7927,8	830,2	1099,2	9858,2
vm_pmfs	test_atime	FPT-SKIP	1	2	2	0	8123,6	819,6	703,4	9647,4
vm_pmfs	test_chmod	FPT-SKIP	5	7	2	0	7887,6	823	2382,6	11094,4
vm_pmfs	test_chown	FPT-SKIP	5	7	2	0	7971,8	824,8	2375,8	11173,4
vm_pmfs	test_ctime-mtime	FPT-SKIP	14	18,2	3	0	7987,4	928,8	6395,8	15313,2
vm_pmfs	test_hello-world	FPT-SKIP	15,4	20	3	0	5903,6	959,6	6661,2	13525,8
vm_pmfs	test_link-hard	FPT-SKIP	13,2	18,2	3	0	7967,6	895,6	6086	14949,8
vm_pmfs	test_link-sym	FPT-SKIP	10	16	2	0	5916,2	958,2	5448,4	12323,8
vm_pmfs	test_mkdir-rmdir	FPT-SKIP	23	34	9	0	7067,6	1073,4	11577,6	19719,8

Continued on next page

Table A.2 – continued from previous page

VM	Test	Tech	Fences	CIs	SSs	FR	Trace (ms)	CI (ms)	SS (ms)	Total (ms)
vm_pmfs	test_rename	FPT-SKIP	17	24,8	8	0	5953,8	980,2	8189	15123,8
vm_pmfs	test_rename-dir	FPT-SKIP	35,4	47	5	0	6029,2	1181,6	16589,2	23801
vm_pmfs	test_rename-long-name	FPT-SKIP	9	12	2	0	5936,8	937,4	4209,8	11085
vm_pmfs	test_touch	FPT-SKIP	14	19	3	0	7892,4	915	6316,4	15125,2
vm_pmfs	test_touch-long-name	FPT-SKIP	20,8	26,8	4	0	7965,4	985,6	9089,2	18041,2
vm_pmfs	test_unlink	FPT-SKIP	13,4	19	8	0	5924,4	931,6	6230,8	13087,8
vm_pmfs	test_update-middle	FPT-SKIP	2	2	2	0	5987,4	851,6	717	7557,4

A.2 Trace Analysis

Table A.3: System 1 TA Average

Test	VM	TA Bugs	TA Entries	Total (ms)
test_append	vm_nova	2	66	156,8
test_atime	vm_nova	2	6	157,2
test_chmod	vm_nova	4	79	160
test_chown	vm_nova	4	79	158,2
test_ctime-mtime	vm_nova	13	172	155,2
test_hello-world	vm_nova	15	4375	523,6
test_link-hard	vm_nova	13	291	156,6
test_link-sym	vm_nova	15	740	165,4
test_mkdir-rmdir	vm_nova	24	347	156,4
test_rename	vm_nova	15	347	160,2
test_rename-dir	vm_nova	40	4810	517
test_rename-long-name	vm_nova	11	305	157,4
test_touch	vm_nova	14	217	157,2
test_touch-long-name	vm_nova	17	4530	521,8
test_unlink	vm_nova	9	159	157,2
test_update-middle	vm_nova	3	65	155,2
test_append	vm_nova-protection	1	105	294
test_atime	vm_nova-protection	3	23	297
test_chmod	vm_nova-protection	3	171	293,2
test_chown	vm_nova-protection	3	171	295,8
test_ctime-mtime	vm_nova-protection	15	272	293,4
test_hello-world	vm_nova-protection	18	4686	658,6
test_link-hard	vm_nova-protection	10	443	296,2
test_link-sym	vm_nova-protection	17	977	303,2
test_mkdir-rmdir	vm_nova-protection	30	654	294,8
test_rename	vm_nova-protection	16	544	299,6
test_rename-dir	vm_nova-protection	49	5515	652,8
test_rename-long-name	vm_nova-protection	9	539	299,4
test_touch	vm_nova-protection	17	452	295,6
test_touch-long-name	vm_nova-protection	21	5007	658,6
test_unlink	vm_nova-protection	9	243	296
test_update-middle	vm_nova-protection	2	104	298,2
test_append	vm_pmfs	1	18	236,4

Continued on next page

Table A.3 – continued from previous page

Test	VM	TA Bugs	TA Entries	Total (ms)
test_atime	vm_pmfs	0	3	239,2
test_chmod	vm_pmfs	3	51	241,6
test_chown	vm_pmfs	3	51	240
test_ctime-mtime	vm_pmfs	7,2	97,6	241,8
test_hello-world	vm_pmfs	11,4	646,4	242
test_link-hard	vm_pmfs	9	127,4	240,4
test_link-sym	vm_pmfs	8	113	239
test_mkdir-rmdir	vm_pmfs	13	207	237
test_rename	vm_pmfs	11,8	139,8	242,2
test_rename-dir	vm_pmfs	26	856,8	245,8
test_rename-long-name	vm_pmfs	9	179	238,6
test_touch	vm_pmfs	9	145	238
test_touch-long-name	vm_pmfs	14,4	759,8	241,8
test_unlink	vm_pmfs	7,2	97,4	239,2
test_update-middle	vm_pmfs	1	11	236

Table A.4: System 1 TA-NoFPT Average

Test	VM	TA Bugs	TA Entries	Total (ms)
test_append	vm_nova	5	66	179,6
test_atime	vm_nova	3	6	176
test_chmod	vm_nova	54	79	180,2
test_chown	vm_nova	54	79	178,2
test_ctime-mtime	vm_nova	50	172	175,6
test_hello-world	vm_nova	126	4375	543,2
test_link-hard	vm_nova	98	291	178,6
test_link-sym	vm_nova	129	740	185,8
test_mkdir-rmdir	vm_nova	114	347	176,8
test_rename	vm_nova	126	347	180,6
test_rename-dir	vm_nova	288	4810	557,4
test_rename-long-name	vm_nova	178	305	180
test_touch	vm_nova	108	217	175,4
test_touch-long-name	vm_nova	237	4530	564,8
test_unlink	vm_nova	45	159	177,6

Continued on next page

Table A.4 – continued from previous page

Test	VM	TA Bugs	TA Entries	Total (ms)
test_update-middle	vm_nova	8	65	174,8
test_append	vm_nova-protection	4	105	333,8
test_atime	vm_nova-protection	3	23	336,2
test_chmod	vm_nova-protection	110	171	334,4
test_chown	vm_nova-protection	110	171	332
test_ctime-mtime	vm_nova-protection	91	272	332
test_hello-world	vm_nova-protection	241	4686	700,4
test_link-hard	vm_nova-protection	184	443	331,4
test_link-sym	vm_nova-protection	241	977	345,6
test_mkdir-rmdir	vm_nova-protection	225	654	330
test_rename	vm_nova-protection	254	544	338,4
test_rename-dir	vm_nova-protection	566	5515	702,8
test_rename-long-name	vm_nova-protection	342	539	338,2
test_touch	vm_nova-protection	216	452	335,8
test_touch-long-name	vm_nova-protection	471	5007	709,4
test_unlink	vm_nova-protection	84	243	331
test_update-middle	vm_nova-protection	7	104	338,8
test_append	vm_pmfs	1,2	18,2	272,2
test_atime	vm_pmfs	0	3	275,8
test_chmod	vm_pmfs	5	51	277
test_chown	vm_pmfs	5,2	51,2	276
test_ctime-mtime	vm_pmfs	9	97,2	273,8
test_hello-world	vm_pmfs	16,2	646,2	277
test_link-hard	vm_pmfs	11	127,2	274
test_link-sym	vm_pmfs	12	113	277,2
test_mkdir-rmdir	vm_pmfs	20	207	276,2
test_rename	vm_pmfs	16,8	139,8	281
test_rename-dir	vm_pmfs	38,8	857	280,4
test_rename-long-name	vm_pmfs	12	179	275
test_touch	vm_pmfs	13	145	276,8
test_touch-long-name	vm_pmfs	22,4	759,8	283,4
test_unlink	vm_pmfs	8,8	97,2	275,4
test_update-middle	vm_pmfs	1	11	273,6

References

[1] Brendan Dolan-Gavitt et al. "Repeatable Reverse Engineering with PANDA." In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. PPREW-5: Program Protection and Reverse Engineering Workshop. Los Angeles CA USA: ACM, Dec. 8, 2015, pp. 1–11. ISBN: 978-1-4503-3642-0. DOI: 10.1145/2843859.2843867. URL: https://dl.acm.org/doi/10.1145/2843859.2843867 (visited on 06/12/2023).

[2] Subramanya R. Dulloor et al. "System Software for Persistent Memory." In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. New York, NY, USA: Association for Computing Machinery, Apr. 14, 2014, pp. 1–15. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592814. URL: https://doi.org/10.1145/2592798.2592814 (visited on 06/05/2023).

[3] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. "Mumak: Efficient and Black-Box Bug Detection for Persistent Memory." In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23: Eighteenth European Conference on Computer Systems. Rome Italy: ACM, May 8, 2023, pp. 734–750. ISBN: 978-1-4503-9487-1. DOI: 10.1145/3552326.3587447. URL: https://dl.acm.org/doi/10.1145/3552326.3587447 (visited on 05/25/2023).

[4] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. *Partial Flush Handling on Mumak · Task3r/Mumak*. GitHub. 2023. URL: https://github.com/task3r/mumak/blob/%20bca1662/instrumentation/src/trace.hpp#L204-L215 (visited on 10/30/2023).

[5] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. *Removed Implicit Flush on Mumak A · Task3r/Mumak*. GitHub. 2023. URL: https://github.com/task3r/mumak/blob/bca1662/instrumentation/src/trace.hpp#L221-L227 (visited on 10/30/2023).

[6] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. *Removed Implicit Flush on Mumak B · Task3r/Mumak*. GitHub. 2023. URL: https://github.com/

task3r/mumak/blob/bca1662/instrumentation/src/trace.hpp#L269-L271 (visited on 10/30/2023).

[7] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7." In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (Jan. 2018), pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[8] *Intel Optane DCPMM UTH DDR T Protocol.* ServeTheHome. URL: https://www.servethehome.com/2nd-gen-intel-xeon-scalable-launch-cascade-lake-details-and-analysis/intel-optane-dcpmm-uth-ddr-t-protocol/ (visited on 06/11/2023).

[9] Joseph Izraelevitz et al. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module.* Aug. 9, 2019. DOI: 10.48550/arXiv.1903.05714. arXiv: 1903.05714 [cs]. URL: http://arxiv.org/abs/1903.05714 (visited on 06/05/2023). preprint.

[10] Rohan Kadekodi et al. "SplitFS: Reducing Software Overhead in File Systems for Persistent Memory." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* SOSP '19. New York, NY, USA: Association for Computing Machinery, Oct. 27, 2019, pp. 494–508. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359631. URL: https://dl.acm.org/doi/10.1145/3341301.3359631 (visited on 06/12/2023).

[11] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. "Vinter: Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems." In: ().

[12] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. "Vinter: Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems." In: (). DOI: 10.5281/zenodo.6626098. URL: https://zenodo.org/records/6626098 (visited on 10/26/2023).

[13] *Kernel Probes (Kprobes) — The Linux Kernel Documentation.* URL: https://www.kernel.org/doc/html/latest/trace/kprobes.html (visited on 06/12/2023).

[14] Seulbae Kim et al. "Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* SOSP '19: ACM SIGOPS 27th Symposium on Operating Systems Principles. Huntsville Ontario Canada: ACM, Oct. 27, 2019, pp. 147–161. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359662. URL: https://dl.acm.org/doi/10.1145/3341301.3359662 (visited on 06/07/2023).

[15] Philip Lantz et al. "Yat: A Validation Framework for Persistent Memory Software." In: 2014 USENIX Annual Technical Conference (USENIX ATC 14). 2014, pp. 433–438. ISBN: 978-1-931971-10-2. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz` (visited on 06/05/2023).

[16] Ryan Latture. *Json2table: Convert JSON to an HTML Table*. Version 1.1.5. URL: `https://github.com/latture/json2table` (visited on 10/23/2023).

[17] *Learning Rust With Entirely Too Many Linked Lists*. URL: `https://rust-unofficial.github.io/too-many-lists/` (visited on 10/22/2023).

[18] Hayley LeBlanc et al. "Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems." In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. New York, NY, USA: Association for Computing Machinery, May 8, 2023, pp. 718–733. ISBN: 978-1-4503-9487-1. DOI: `10.1145/3552326.3567498`. URL: `https://dl.acm.org/doi/10.1145/3552326.3567498` (visited on 05/25/2023).

[19] Jayashree Mohan et al. "CrashMonkey and ACE: Systematically Testing File-System Crash Consistency." In: *ACM Transactions on Storage* 15.2 (Apr. 20, 2019), 14:1–14:34. ISSN: 1553-3077. DOI: `10.1145/3320275`. URL: `https://dl.acm.org/doi/10.1145/3320275` (visited on 06/03/2023).

[20] Thomas-Christian Oder. *Partial Flush Handling in Vinter's Trace Analyzer · Myself5/Vinter*. 2023. URL: `https://github.com/Myself5/vinter/blob/thesis/vinter_rust/vinter_report/src/lib.rs#L627-L632` (visited on 10/30/2023).

[21] Thomas-Christian Oder. *Vinter_trace2img Testing: Skip FPT Check for Efficiency Comparision · Myself5/Vinter*. GitHub. 2023. URL: `https://github.com/Myself5/vinter/commit/001be70` (visited on 11/02/2023).

[22] Thomas-Christian Oder et al. *Source Changes for "Fast Persistent Memory Crash Consistency Analysis Based on Virtual Machines" · Myself5/Vinter*. 2023. URL: `https://github.com/Myself5/vinter/tree/thesis` (visited on 10/30/2023).

[23] *Pin - A Dynamic Binary Instrumentation Tool*. Intel. URL: `https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html` (visited on 05/25/2023).

[24] *R3bl_rs_utils 0.9.10 - Docs.Rs*. URL: `https://docs.rs/crate/r3bl_rs_utils/latest` (visited on 10/22/2023).

[25] *Syzkaller - Kernel Fuzzer*. Google, June 12, 2023. URL: `https://github.com/google/syzkaller` (visited on 06/12/2023).

[26] *Uprobe-Tracer: Uprobe-based Event Tracing — The Linux Kernel Documentation*. URL: `https://www.kernel.org/doc/html/latest/trace/uprobetracer.html` (visited on 06/12/2023).

[27] Jian Yang et al. "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory." In: 18th USENIX Conference on File and Storage Technologies (FAST 20). 2020, pp. 169–182. ISBN: 978-1-939133-12-0. URL: `https://www.usenix.org/conference/fast20/presentation/yang` (visited on 06/05/2023).