Improvements in Crash Consistency Testing for Persistent Memory File Systems

Lukas Werling
Karlsruhe Institute of Technology
Germany

Thomas-Christian Oder
Karlsruhe Institute of Technology
Germany

Lucas Wäldele
Karlsruhe Institute of Technology
Germany

Daniel Ritz
Karlsruhe Institute of Technology
Germany

Frank Bellosa
Karlsruhe Institute of Technology
Germany

ABSTRACT

Achieving crash consistency in persistent memory (PM) applications is difficult due to the need for fine-granular cache flushes and memory fences. File systems are no exception. Previous works have found numerous bugs in PM file systems, but crash consistency testing as part of file system development is still rare. With this paper, we make crash consistency testing easier by improving Vinter, a crash consistency testing tool for file systems based on virtual machines. We introduce support for cross-media file systems with NVMe and PM that are not covered by other testing tools. To speed up testing, we add an alternative algorithm for simulating crashes that focuses on logic bugs rather than misuse of PM primitives.

KEYWORDS

crash consistency, file systems, persistent memory, NVMe, cross-media, bug detection, testing

1 INTRODUCTION

Persistent memory (PM) allows direct access to storage with load and store instructions. Due to volatile caches and out-of-order execution, special care is needed to ensure that PM contents are consistent at all times. Otherwise, a crash (e.g., after a power failure) could lead to data loss. Applications use special persistency primitives such as non-temporal stores, cache flushes, and memory fences to control the order in which modifications reach PM. Consequently, applications need adaptation to work correctly with directly-accessed PM.

As an alternative, PM file systems can provide access to PM over the traditional file system interface. They thus allow high-performance PM access for unmodified applications.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. DOI: https://doi.org/10.18420/fgbs2024f-01. FGBS '24, March 14-15, 2024, Bochum, Germany

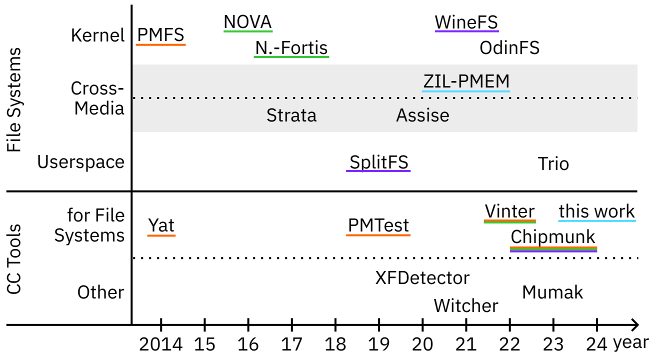

Figure 1: A timeline of selected PM file systems and crash consistency testing tools. The underlines indicate which file systems were tested by the respective tools.

Starting with PMFS in 2014, a large number of PM file systems have appeared over the last years [5, 9, 12, 13, 15, 23–29].

Since the PM persistency primitives do not change the visible runtime behavior of the application, we need tools to verify their correct use. Previous works have proposed a variety of such tools [7, 10, 11, 14, 16–19, 21]. Due to challenges in tracing kernel code, only a small subset of these tools (Yat [16], PMTest [19], Vinter [14], and Chipmunk [17]) target PM file systems. All of these tools found several bugs in the file systems they tested. This highlights the importance of automated crash consistency testing to verify crash consistency behavior.

Figure 1 shows an overview of crash consistency testing tools and the file systems they analyzed. We have a good understanding of the crash consistency behavior of these tested file systems. However, we observe that crash consistency testing as part of file system development is still rare. Given that recent crash consistency testing tools found a multitude of issues in the file systems they tested, there is a high likelihood of undiscovered bugs in newly developed systems. Possible reasons for reluctance to use crash consistency testing tools are the large time investment necessary

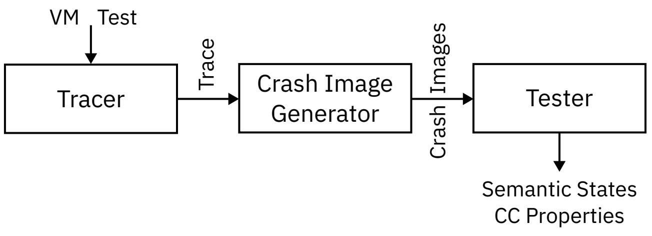

Figure 2: Overview of Vinter's crash consistency testing pipeline.

due to slow analysis tools, as well as missing features for analyzing specialized systems.

With our work, we make crash consistency testing easier by introducing improvements to Vinter [14]. We add support for cross-media file systems that store data on NVMe in addition to PM (Section 3), and we improve the overall testing speed (Section 4).

2 BACKGROUND AND RELATED WORK

Since our work in this paper builds on top of Vinter, we first take a closer look at it. We then compare Vinter to newer crash consistency testing tools.

2.1 Vinter

Vinter [14] is a crash consistency testing tool for PM file systems. It performs black-box testing of full systems running in a virtual machine and can automatically detect crash consistency properties including *single final state* and atomicity. Its crash consistency testing pipeline has three stages, as shown in Figure 2: tracer, crash image generator, and tester.

Vinter's **tracer** uses PANDA [8], a platform for dynamic analysis based on QEMU with binary translation. It runs a test in a virtual machine and configures PANDA to hook all memory accesses as well as PM persistence primitives such as cache flush and memory fence instructions. After filtering relevant events, a thread asynchronously serializes and compresses these events into a trace file.

After tracing finishes, the **crash image generator** reads the trace, simulating PM and cache contents. At every memory fence with active writes, the crash image generator stores possible contents of PM as if the virtual machine would crash at that point. The images include one with no in-flight writes, another with all writes, and more images with subsets of writes. Since applications can perform an arbitrary amount of writes between memory fences, this can easily lead to a combinatorial explosion of possible states. Vinter employs a heuristic based on read accesses by the recovery to reduce this search space.

Finally, the **tester** loads each crash image into a new virtual machine, runs the recovery process, and extracts the semantic state from the image. For a file system, the semantic state is a dump of all files and directories with their metadata and contents. From the semantic states, the tester can automatically derive two crash consistency properties. First, single final state means that at the end of the test case, there is only one unique semantic state. Second, an operation is atomic if there are exactly two unique semantic states in total.

2.2 Newer Crash Consistency Testing Tools

We now take a short look at Chipmunk [17] and Mumak [11], two crash consistency testing tools for PM that were published after Vinter. Both tools use a similar testing pipeline. Rather than using manually-written file system test cases as in Vinter, Chipmunk can dynamically generate them [17].

Mumak's tracer uses binary instrumentation, allowing black box testing similar to Vinter, but only with user space binaries [11]. Chipmunk opts for manual instrumentation of relevant function calls using Linux Kprobes and Uprobes [17]. It thus requires detailed knowledge of the tested application to instrument the correct functions.

Crash image generation in Chipmunk is comparable to Vinter. Instead of a heuristic based on recovery read accesses, Chipmunk coalesces stores performed by one function call (e.g., a `memcpy` function) to a single write [17]. Mumak's crash image generation is focused on low analysis time by reducing the amount of images. We take a closer look at it in Section 4.

Chipmunk is based on CrashMonkey [20], a crash consistency testing tool for block-based file systems. However, it is not clear whether Chipmunk retains the ability to trace block devices in addition to PM for testing cross-media file systems, since none of their tested file systems require block devices.

3 TESTING CROSS-MEDIA PM FILE SYSTEMS

Cross-media PM file systems are file systems that use additional types of storage devices such as SSDs and hard disks. The different storage media are usually arranged in tiers, with fast but small PM serving latency-critical accesses and block device tiers providing capacity for long-term storage at higher latency.

For cross-media file systems, analyzing just the PM portion with a tool like Vinter is not sufficient. Crash images need to include the full state of the file system, including PM and block devices. The access mode and crash consistency semantics of block devices are very different from PM. Access to block devices happens over asynchronous interfaces such as NVMe. Instead of cache line flushes and memory barriers, NVMe specifies a flush command that commits previous writes to non-volatile memory [4].

In this section, we extend Vinter to support cross-media PM file systems. The resulting tool *Permanent* is capable of testing cross-media file systems that use PM and an NVMe block device.

3.1 Approach

To enable testing cross-media file systems, we need to modify all Vinter components. We discuss changes to the tracer, the crash image generator, and the tester in sequence.

The **tracer** needs to provide an NVMe device to the virtual machine, and trace all relevant commands issued to it. Relevant commands are the NVMe write command that instructs the SSD to write one or more blocks [2], and the NVMe flush command that flushes all preceding write commands from any volatile on-device buffers to persistent storage [4]. The NVMe flush thus acts similar to a global cache line flush and a memory fence for PM. We assume an atomic block size of 512 bytes and split larger write commands to that size. The tracer captures a combined trace with PM and NVMe events in order.

The **crash image generator** receives the trace and produces combined crash images (PM, NVMe). Cross-device reordering is the primary challenge. Figure 3 shows an example trace with two PM events and one NVMe event. If we do not allow reordering between devices, we end up with the three crash images as shown. With cross-device reordering, a fourth image ({ }, {1}) would appear. Unrestricted cross-device reordering leads to an explosion of possible states and thus is not feasable. We argue that disallowing such reordering better matches the behavior of the hardware, and thus implement this model in Permanent. NVMe command submission on x86 involves a store to uncacheable (UC) memory-mapped I/O addresses, and command completion is signalled with an interrupt. Since both of these are serializing events [1], reordering PM accesses across them is not possible.

To generate combined crash images, we first consider each device independently. Crash image generation for both devices happens at each PM fence with pending PM cache line flushes, and at each NVMe flush with pending NVMe writes. PM crash image generation then works like in Vinter. For NVMe crash images, we allow arbitrary reordering of all NVMe writes since the previous flush. We found that Vinter's heuristic does not sufficiently reduce the search space for file systems such as ext4. We thus only use random combinations of NVMe writes with a cutoff.

To combine the crash images from the two device types, Permanent stores an index containing the set of PM crash images and the set of NVMe crash images generated at that point. Again, this combination could lead to a combinatorial explosion of possible states, so we use a limited number of random subsets.

Finally, the **tester** loads combined crash images to recover their semantic state. Other than the combined loading, its functionality stays the same as in Vinter.

3.2 Implementation

We implemented Permanent based on Vinter's Rust implementation. Its source code is available at https://github.com/KIT-OSGroup/permanent.

The main challenge was the tracer, which required a full rewrite. Vinter's tracer is based on PANDA [8], which itself is based on an older QEMU version that does not include a virtual NVMe device. For this reason, we reimplemented PM tracing on top of vanilla QEMU with its TCG plugin capabilities [6]. TCG plugins allow registering callbacks for certain instructions, including memory accesses. To work around some limitations in TCG plugins, we had to patch QEMU to provide functions for reading guest memory to our TCG plugin. Since register values are only available at the end of basic blocks, we additionally patched the code generation to pass the address of memory flush instructions via a memory access callback. Similarly, we implemented hypercalls as a memory access combined with a signalling `mov` instruction with a special immediate value.

We implemented NVMe tracing by introducing callbacks at relevant points in the NVMe emulation code. A single NVMe command triggers multiple callbacks with a subset of the information such as the command's type and the data to be written. All events from these NVMe and PM callbacks are inserted into a queue. A worker thread then asynchronously reassembles NVMe events to full commands, serializes the NVMe commands and PM events, and writes them out to the trace file.

3.3 Evaluation

To demonstrate Permanent's functionality, we use it to analyze ZIL-PMEM [22]. ZIL-PMEM is an extension to OpenZFS, a file system for block devices. ZIL-PMEM stores synchronous file system accesses in a ring buffer structure on PM, allowing acknowledgement of such accesses with low latency. Once ZFS asynchronously commits the access to block storage, the corresponding entry may be removed from PM.

We use Vinter's test cases for our analysis. The resulting semantic states mostly meet our expectations for the respective tests. Some of the tests yielded unexpected states because ZFS updates timestamps on files that it modifies during recovery, which we do not consider a problem. We therefore did not discover any bugs in ZIL-PMEM or ZFS.

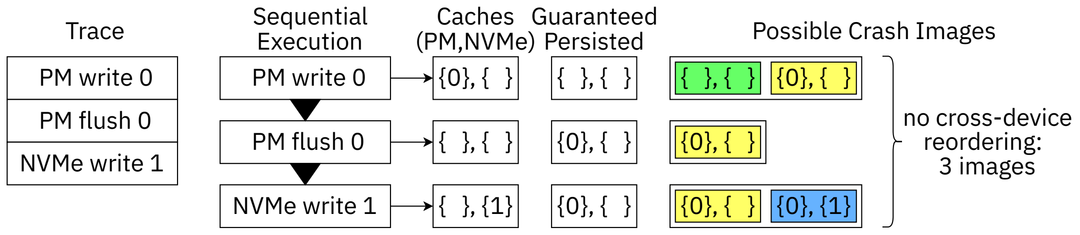

Figure 3: Example for crash image generation without cross-device reordering, as implemented by Permanent.

3.4 Discussion

Our current prototype of Permanent has a number of limitations. A couple of features from Vinter are missing, including stack traces for trace entries, evaluating the heuristic for PM crash image generation, and speeding up state extraction in the tester using VM snapshots. We do not expect obstacles in adding these features.

Permanent's PM tracing is significantly slower than Vinter's tracer. We expect a large potential for improvement by adopting PANDA's approach to memory access tracing [8].

We only applied Permanent to ZIL-PMEM so far. This is partly because there is no source code available to other cross-media file systems. For Strata [15] and Assise [5], two user space cross-media file systems, LeBlanc et al. note that the prototypes of these file systems do not currently support recovery from arbitrary crashes [17].

Finally, Permanent can only test file system interaction with QEMU's NVMe implementation. Since the NVMe standard leaves some freedom to implementations, the behavior of real SSDs might differ. In particular, the virtual NVMe device always signals completion immediately and in order of submission. Permanent thus cannot detect bugs from file systems that do not handle delayed or reordered completions correctly.

4 FAST TESTING FOR LOGIC BUGS

A core problem with crash consistency testing is the amount of crash images generated from the trace. Programs can write an arbitrary amount of data between memory fences, leading to an explosion of possible crash images. Vinter uses a heuristic based on read accesses done by the recovery routine [14]. Tracing the recovery routine for the heuristic takes additional time, though, and might not sufficiently cut down the number of writes.

Mumak [11] takes an approach that prioritizes speed of the analysis over coverage of possible crash states. By generating crash images without considering partial cache flushes and deduplicating crash image generation points by stack trace, Mumak can find *logic bugs* very quickly. Misuse of PM primitives (e.g., missing cache line flushes) cannot reliably be detected with such crash images. However, LeBlanc et al.

	Vinter	**Mumak**
where	memory fences	cache flushes and memory fences
when	always	once per unique stack trace
contents	no in-flight stores, all stores, subset of stores with heuristic	all stores

Figure 4: Differences in the crash image generation algorithms of Vinter [14] and Mumak [11].

have shown that logic bugs make up the majority of all bugs they detected in PM file systems [17]. Thus, fast analysis for logic bugs is a useful capability for a file system crash consistency testing tool. Since Mumak can only analyze user space software, it cannot analyze most file systems with kernel components.

In this section, we describe our work of bringing fast Mumak-style analysis to Vinter. We design and implement *Vinter-FPT*, which replaces Vinter's crash image generation algorithm. Additionally, we bring Mumak's trace analysis to Vinter which provides hints for certain types of PM primitives misuse that Vinter-FPT cannot otherwise detect. Vinter-FPT's source code is available at https://github.com/KIT-OSGroup/Vinter-FPT.

4.1 Crash Image Generation

As outlined in Section 2, the testing pipelines of Mumak and Vinter are very similar. We can thus keep Vinter's tracer and tester components in Vinter-FPT without modifications and only need to introduce a new algorithm to the crash image generator.

The main differences from Vinter's original algorithm are *where* and *when* crash images are generated and their contents, summarized in Figure 4. Consequently, we modified Vinter to also generate crash images at cache flushes with unpersisted stores. Instead of using Vinter's heuristic to generate multiple images with a subset of all unpersisted stores, Vinter-FPT always generates exactly two images: one that includes all stores up to this point (like Mumak), and another that only includes stores that are fully persisted (i.e., after a flush and fence).

Finally, Vinter-FPT only generates crash images once per unique stack trace at a particular cache flush or memory fence instruction with unpersisted stores. We ported Mumak's *failure point tree* [11] for this purpose. The tree efficiently stores the stack traces at each crash image generation point. We skip crash image generation if the stack trace we try to insert is already present. This strategy of building and querying the failure point tree simultaneously differs slightly from Mumak, which builds the tree during tracing [11]. We combine these steps during crash image generation to avoid modifying Vinter's tracer.

Note that the tracer in Vinter collects stack traces only as optional metadata for debugging, since Vinter's original crash image generator does not require stack traces. Stack traces must be captured for deduplication with the failure point tree, resulting in some overhead during tracing compared to original Vinter.

4.2 Trace Analysis

As an addition to crash image generation, Mumak implements a trace analyzer that searches the trace for patterns that indicate misuse of PM persistency primitives [11]. Although trace analysis cannot confirm bugs at the same level as the testing pipeline, it can hint at some problems such as ommitted cache flush instructions that the fast crash image generation algorithm misses.

Our implementation of trace analysis closely follows Mumak's design [11]. In order to reduce the amount of reported problems, we introduce a variant of the failure point tree for deduplication. Using a flag for each problem in the leaves of the tree, we only output a report the first time we encounter it for a particular stack trace.

4.3 Results

We repeat the analysis of PMFS [9], NOVA [24], and NOVA-Fortis [25] with Vinter's 16 original test cases [14]. We answer the following questions: Does the new crash image generation algorithm result in a faster overall analysis? Are we able to find the same bugs as original Vinter?

4.3.1 Performance. We run all tests on the three file systems and track the runtime of the three components tracer, crash image generator, and tester. Our test system has an AMD Ryzen Threadripper 3970X CPU running at 4.1 GHz. We compare three configurations. *Vinter* is the original crash image generation. *No dedup* generates crash images as described above, but does not deduplicate by stack traces. *Full* includes deduplication.

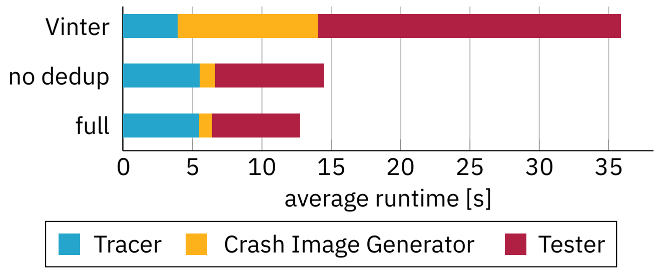

Figure 5: Average runtime of Vinter over all file system tests compared with Vinter-FPT in two variants.

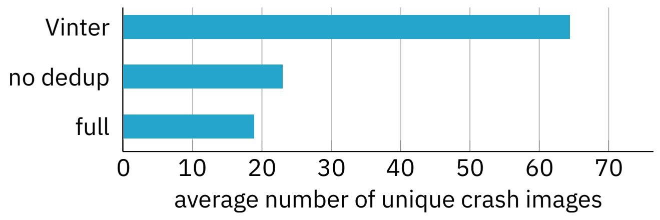

Figure 6: Average number of unique crash images generated by Vinter and Vinter-FPT variants over all file system tests.

Figure 5 shows the average runtime over all tests, and Figure 6 the average number of generated crash images. We can see an overall reduction of runtime from *Vinter* to *full* of 64.5%. The biggest relative reduction is from the crash image generator that takes less than 10% of the original runtime. This result is expected, since original Vinter spends more than half of the crash image generator runtime evaluating its heuristic [14] and generates more than three times the amount of crash images.

The runtime of the tracer increases by 28.1% on average when capturing stack traces. For some tests with very few (< 5) crash images, this leads to a small increase in overall runtime. Additionally, we observe some tests where the number of crash images and thus time spent in the tracer increases, as Vinter-FPT generates crash images at more trace entries than original Vinter.

Deduplication with the failure point tree yields an overall runtime improvement of 18.3% on average. The crash image generator spends 15.6% less time generating and then deduplicating images. The number of unique images decreases by around 4 on average, improving runtime of the tester by 19.6%.

4.3.2 Completeness. We now compare the test results from original Vinter with those from Vinter-FPT. We summarize our observations in Figure 7. In more than half of the tests, the resulting states are identical. Although Vinter-FPT finds fewer states than Vinter in 14 tests, the resulting properties

Observation	# tests
Identical states in both	25
Failed recovery in Vinter, but not Vinter-FPT	3
Fewer states in Vinter-FPT, same result	14
Fewer states, **wrongly assumed atomic**	6

Figure 7: Evaluation of test results from Vinter and Vinter-FPT. Vinter-FPT misses atomicity bugs in 6 out of 48 tests.

(single final state, atomicity) match. In three tests, Vinter-FPT does not find states whose recovery fails, which Vinter considers a separate type of bug. However, Vinter-FPT still flags these tests due to atomicity violations.

Finally, six tests remain where Vinter-FPT finds fewer semantic states than Vinter and thus wrongly considers the operation atomic. For these tests, we take a look at the output of the trace analyzer. It reports unordered flushes for three tests, and a missing flush for two other tests. Only for one remaining test, there is no hint at any problem.

4.4 Discussion

Vinter-FPT meets expectations. It significantly reduces analysis time for most test cases. Only in very short tests where Vinter already generates few crash images, runtime increases slightly from capturing stack traces or generating more images. The generated crash images allowed reproduction of the bugs that Vinter found in 39 of 48 test cases, and found less severe issues than Vinter in three more test cases. This is close to Mumak's claim of a 90% bug coverage [11].

Although we could see that trace analysis can help to close some gaps of the fast crash image generator, we found that its reports are generally less helpful than reports from crash images. For some test cases, it generated a large amount of reports, even after deduplication. Reports of unordered flushes (i.e., a fence acting on multiple flush instructions) were the only hints at problems for three tests where Vinter-FPT's crash image generation failed. This pattern does not necessarily indicate a bug, so confirmation with slower, but more comprehensive crash image generation like Vinter's is necessary.

5 FUTURE WORK

Improving tracing. As we have seen in Figure 5, Vinter's tracing step makes up a large portion of the analysis time. Function-based tracing as in Chipmunk [17] is significantly faster, but requires manual, error-prone input. The primary issue is that dynamic binary translation needs to translate all instructions and hook all memory accesses, even though only a small subset of them access PM. As an alternative, we are investigating patching only relevant memory access instruction at runtime. Using memory protection keys (e.g., Intel MPK [1]), selectively trapping access to PM is possible. We can then patch the trapping instruction, trace the access, and repeat the original instruction with adjusted memory protection keys to allow access. Further executions of the patched instructions will not trap, allowing fast tracing.

CXL memory crash consistency. Compute Express Link (CXL) [3] allows devices to provide persistent memory to the system. CXL provides a unique opportunity to gain more insight into the system's crash consistency semantics. With a custom CXL device (e.g., an FPGA), we can trace PM accesses at the device. By cross-referencing the device trace with a CPU trace, we can verify the accuracy of the underlying models of crash consistency testing tools.

6 CONCLUSION

Crash consistency testing, especially for file systems, is an important tool for writing correct PM software. We have introduced two improvements to Vinter, a crash consistency testing tool based on virtual machines.

Permanent extends Vinter to support cross-media file systems that store data on NVMe in addition to PM. It traces accesses to a virtual NVMe device and creates combined (PM, NVMe) crash images. We have successfully used Permanent to test the cross-media file system ZFS with ZIL-PMEM.

Vinter-FPT speeds up Vinter's crash image generation by focusing on logic bugs. We have demonstrated that Vinter-FPT can reduce the runtime by more than half while still finding most of the bugs in the tested file systems.

REFERENCES

[1] 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D.* Technical Report. 4670 pages.

[2] 2023. NVM Express NVM Command Set Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-1.0d-2023.12.28-Ratified.pdf

[3] 2024. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://computeexpresslink.org

[4] 2024. NVM Express Base Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0d-2024.01.11-Ratified.pdf

[5] Thomas E Anderson, Simon Peter, Marco Canini, Jongyul Kim, Dejan Kostic, Youngjin Kwon, Waleed Reda, Henry N Schuh, and Emmett Witchel. 2022. Assise: Performance and Availability via Client-local NVM in a Distributed File System. *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2022), 1011–1027. https://www.usenix.org/conference/osdi20/presentation/anderson

[6] The QEMU Project Developers. 2023. QEMU TCG Plugins. https://www.qemu.org/docs/master/devel/tcg-plugins.html

[7] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, flexible, and comprehensive bug detection for persistent memory programs. In

Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, Virtual USA, 503–516. https://doi.org/10.1145/3445814.3446744

[8] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM, Los Angeles CA USA, 1–11. https://doi.org/10.1145/2843859.2843867

[9] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, 15:1–15:15. https://doi.org/10.1145/2592798.2592814

[10] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, Virtual Event Germany, 100–115. https://doi.org/10.1145/3477132.3483556

[11] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. 2023. Mumak: Efficient and Black-Box Bug Detection for Persistent Memory. In *Proceedings of the Eighteenth European Conference on Computer Systems*. ACM, Rome Italy, 734–750. https://doi.org/10.1145/3552326.3587447

[12] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. ACM, Virtual Event Germany, 804–818. https://doi.org/10.1145/3477132.3483567

[13] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, Huntsville Ontario Canada, 494–508. https://doi.org/10.1145/3341301.3359631

[14] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. 2022. Vinter: Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 933–950. https://www.usenix.org/conference/atc22/presentation/werling

[15] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, Shanghai China, 460–477. https://doi.org/10.1145/3132747.3132770

[16] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 433–438. https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

[17] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. 2023. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 718–733. https://doi.org/10.1145/3552326.3567498

[18] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 1187–1202. https://doi.org/10.1145/3373376.3378452

[19] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Providence RI USA, 411–425. https://doi.org/10.1145/3297858.3304015

[20] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Oct. 2018), 33–50. https://www.usenix.org/conference/osdi18/presentation/mohan

[21] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1047–1064. https://www.usenix.org/conference/osdi20/presentation/neal

[22] Christian Schwarz. 2021. Low-latency synchronous IO for OpenZFS using persistent memory. https://os.itec.kit.edu/downloads/2021_MA_Schwarz_SyncIOForZFS.pdf

[23] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, 14:1–14:14. https://doi.org/10.1145/2592798.2592810

[24] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 323–338. https://www.usenix.org/node/194455

[25] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 478–496. https://doi.org/10.1145/3132747.3132761

[26] Yang Yang, Qiang Cao, Jie Yao, Yuanyuan Dong, and Weikang Kong. 2021. SPMFS: A Scalable Persistent Memory File System on Optane Persistent Memory. In *50th International Conference on Parallel Processing (ICPP 2021)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3472456.3472503

[27] Shengan Zheng, Morteza Hoseinzadeh, Steven Swanson, and Linpeng Huang. 2023. TPFS: A High-Performance Tiered File System for Persistent Memories and Disks. *ACM Transactions on Storage* 19, 2 (May 2023), 1–28. https://doi.org/10.1145/3580280

[28] Shawn Zhong, Chenhao Ye, and Guanzhou Hu Suyan Qu. 2023. MadFS: Per-File Virtualization for Userspace Persistent Memory Filesystems. *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Feb. 2023), 265–280. https://www.usenix.org/conference/fast23/presentation/zhong

[29] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 179–193. https://www.usenix.org/conference/osdi22/presentation/zhou-diyu