Understanding Persistent-memory-related Issues in the Linux Kernel

OM RAMESHWAR GATLA, DUO ZHANG, WEI XU, and MAI ZHENG, Department of Electrical and Computer Engineering, Iowa State University

Persistent memory (PM) technologies have inspired a wide range of PM-based system optimizations. However, building correct PM-based systems is difficult due to the unique characteristics of PM hardware. To better understand the challenges as well as the opportunities to address them, this article presents a comprehensive study of PM-related issues in the Linux kernel. By analyzing 1,553 PM-related kernel patches in depth and conducting experiments on reproducibility and tool extension, we derive multiple insights in terms of PM patch categories, PM bug patterns, consequences, fix strategies, triggering conditions, and remedy solutions. We hope our results could contribute to the development of robust PM-based storage systems.

CCS Concepts: • **Software and its engineering** → **Operating systems**; • **Hardware** → *Memory and dense storage;*

Additional Key Words and Phrases: Persistent memory, kernel patches, bug detection, reliability

ACM Reference format:
Om Rameshwar Gatla, Duo Zhang, Wei Xu, and Mai Zheng. 2023. Understanding Persistent-memory-related Issues in the Linux Kernel. *ACM Trans. Storage* 19, 4, Article 36 (September 2023), 28 pages.
https://doi.org/10.1145/3605946

1 INTRODUCTION

Persistent memory (PM) technologies offer attractive features for developing storage systems and applications. For example, **phase-change memory (PCM)** [95], **spin-transfer torque RAM (STT-RAM)** [62], Intel's infamous Optane DCPMM [10], and the promising vendor-neutral **Compute Express Link–(CXL)** based PM technologies [34, 91] can support byte-granularity accesses with close to DRAM latencies, while also providing durability guarantees. Such new properties have inspired a wide range of PM-based software optimizations [8, 41, 52, 70, 98, 114].

Unfortunately, building correct PM-based software systems is challenging [69, 93]. For example, to ensure persistence, PM writes must be flushed from CPU cache explicitly via specific instructions (e.g., `clflushopt`); to ensure ordering, memory fences must be inserted (e.g., `mfence`). Moreover, to manage PM devices and support PM programming libraries (e.g., PMDK [19]), multiple

This work was supported in part by NSF under grants CNS-1566554, CNS-1855565, CNS-1943204, and a gift from Western Digital/IDEMA.
Author's address: O. R. Gatla, D. Zhang, W. Xu, and M. Zheng, Department of Electrical and Computer Engineering, Iowa State University; emails: {ogatla, duozhang, weixu, mai}@iastate.edu.

1553-3077/2023/09-ART36 $15.00
https://doi.org/10.1145/3605946

operating system (OS) kernel subsystems must be revised (e.g., `dax`, `libnvdimm`). Such complexity could potentially lead to obscure bugs that hurt system reliability and security.

Addressing the challenge above will require cohesive efforts from multiple related directions including PM bug detection [38, 76, 77, 88, 89], PM programming support [19], PM specifications [46], among others. All of these directions will benefit from a better understanding of real-world PM-related bug characteristics.

Many studies have been conducted to understand and guide the improvement of software [39, 43, 54, 68, 78, 80]. For example, Lu et al. [78] studied 5,079 patches of six Linux file systems and derived various patterns of file system evolution; the study has inspired various follow-up research on file systems reliability [51, 86], and the dataset of file system bug patches has been directly used for evaluating the effectiveness of new bug detection tools [86]. While influential, this study does not cover PM-related issues, as the **direct-access (DAX)**, feature of file systems was introduced after this study was performed. More recently, researchers have studied PM related bug cases. For example, Neal et al. [88] studied 63 PM bugs (mostly from the PMDK library [19]) and identified two general patterns of PM misuse. While these existing efforts have generated valuable insights for their targets, they do not cover the potential PM-related issues in the Linux kernel.

In this work, we perform the first comprehensive study on PM-related bugs in the Linux kernel. We focus on the Linux kernel for its prime importance in supporting PM programming [14, 16, 20]. Our study is based on 1,553 PM-related patches committed in Linux between January 2011 and December 2021, spanning over 10 years. For each patch, we carefully examine its purpose and logic, which enables us to gain quantitative insights along multiple dimensions:

First, we observe that a large number of PM patches (38.9%) are for maintenance purpose, and a similar portion (38.4%) are for adding new features or improving the efficiency of existing ones. These two major categories reflect the significant efforts needed to add PM devices to the Linux ecosystem and to keep the kernel well maintained. Meanwhile, a non-negligible portion (22.7%) are bug patches for fixing correctness issues.

Next, we analyze the PM bug patches in depth. We find that the majority of kernel subsystems have been involved in the bug patches (e.g., "`arch`," "`fs`," "`drivers`," "`block`," and "`mm`"), with drivers and file systems being the most "buggy" ones. This reflects the complexity of implementing the PM support correctly in the kernel, especially the `nvdimm` driver and the DAX file system support.

In terms of bug patterns, we find that the classic semantic and concurrency bugs remain pervasive in our dataset (49.7% and 14.8%, respectively), although the root causes are different. Also, many PM bugs are uniquely dependent on hardware (19.0%), which may be caused by misunderstanding of specifications, miscalculation of addresses, and so on. Such bugs may lead to missing devices, inaccessible devices, or even security issues, among others.

In terms of bug fixes, we find that PM bugs tend to require more lines of code to fix compared to non-PM bugs reported in previous studies [78]. Also, 20.8% bugs require modifying multiple kernel subsystems to fix, which implies the complexity. In the extreme cases (0.9%), developers may temporarily "fix" a PM bug by disabling a PM feature, hoping for a major re-work in the future. However, we observe that different PM bugs may be fixed in a similar way by refining the sanity checks.

Moreover, to better understand the conditions for manifesting the issues and help develop effective remedy solutions, we identify a subset of bug patches with relatively complete information and attempt to reproduce them experimentally. We find that configuration parameters in different utilities (e.g., `mkfs` for creating a file system and `ndctl` for managing the `libnvdimm` subsystem) are critically important for manifesting the issues, which suggests that it is necessary to take into account the configuration states when building bug detection tools.

Finally, we look into the potential solutions to address the PM-related issues in our study. We examine multiple representative PM bug detectors [76, 77, 88] and find that they are largely inadequate for addressing the PM bugs in our study. However, a few recently proposed non-PM bug detectors [63, 66, 99, 101] could potentially be applied to detect a great portion of PM bugs if a few common challenges (e.g., precise PM emulation, PM-specific configuration, and workload support) are addressed. To better understand the feasibility of extending existing tools for PM bug detection, we further extend one existing bug detector called Dr.Checker [82] to analyze PM kernel modules. By adding PM-specific modifications, the extended Dr.Checker, which we call Dr.Checker+, can successfully analyze the major PM code paths in the Linux kernel. While the effectiveness is still limited by the capability of the vanilla Dr.Checker, we believe that extending existing tools to make them work for the PM subsystem can be an important first step toward addressing the PM-related challenges exposed in our study.

Note that this article is extended from a conference version [106]. The major changes include (1) collecting and analyzing one new year of PM-related patches (i.e., January to December 2021), (2) conducting reproducibility experiments and identifying manifestation conditions, (3) analyzing more existing tools and extending Dr.Checker for analyzing PM driver modules, and (4) adding background, bug examples, and so on, to make the article more clear and complete. We have released our study results including the dataset and the extended Dr.Checker+ publicly on Git [3]. We hope our study could contribute to the development of effective PM bug detectors and the enhancement of robust PM-based systems.

The rest of the article is organized as follows: Section 2 provides background on the unique hardware and software characteristics of PM devices; Section 3 describes the study methodology; Section 4 presents the overview of PM patches; Section 5 characterizes PM bugs in detail; Section 6 presents our experiments on PM bug reproducibility; Section 7 discusses the implications on bug detection, including our extension to Dr. Checker for analyzing PM subsystem in the Linux kernel; Section 8 discusses related work; and Section 10 concludes the article.

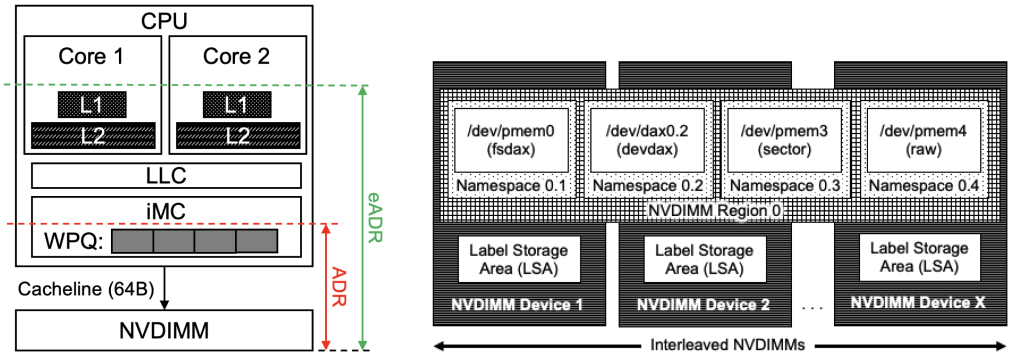
2 BACKGROUND

In this section, we introduce the characteristics of PM hardware and software that may contribute to correctness issues in PM-based systems.

2.1 Non-volatile Memory and PM Device Types

A wide range of **non-volatile memory (NVM)** devices have been proposed and are being developed with different degrees of maturity. To facilitate standardization, the JEDEC specification [11] classifies NVM devices into the following three types based on the method used for persistence:

- NVDIMM-N devices employ both DRAM and Flash modules and is the initial form of PM devices developed. These devices achieve non-volatility by copying contents in the DRAM to the flash modules when the host power is lost using an energy source managed by either the device (e.g., on-chip capacitor) or the host. AGIGA's NVDIMM-N [2] and HPE NVDIMM [7] devices are a few examples.
- NVDIMM-F devices use NAND flash modules over the memory bus as storage medium. Some eamples of this type are SanDisk's ULLtraDIMM [28] and IBM's exFlash DIMM [9]. However, due to their high access latency, these devices have largely been discontinued.
- NVDIMM-P standard encompasses devices that employ new storage technologies such as PCM [95], STT-RAM [62], ReRAM [44], and so on, to achieve persistence. The commercialized Intel Optane DCPMM and the vendor-neutral CXL-based PMs also belong to this category.



(a) PM Persistence Domains (b) Software Abstractions of NVDIMM Subsystem in Linux

Fig. 1. Background of PM hardware and software.

In addition, to facilitate NVM programming, the Storage Networking Industry Association [22] differentiates the concepts of NVM and PM [18]: NVM refers to any type of memory-based, persistent media (including flash memory packaged as solid state disks), while PM refers to a subset of NVM technology with performance characteristics suitable for a load and store programming (e.g., PCM, STT-RAM, and Optane DCPMM). In this article, we follow the definition and study the Linux kernel issues related to such PM devices.

2.2 PM Device Characteristics

PM devices reside on the memory bus along with DRAM devices. Unlike traditional block storage devices that provide a block IO interface to access data, PM devices use memory load and store instructions. These devices typically offer lower access latency and higher endurance compared to flash-based storage devices. For example, STT-RAM and PCM devices may offer latencies in the range of 5–220 ns with an endurance rate around 10^{12} writes per bit [103].

Among existing PM devices, Intel Optane DCPMM has been the most prominent one. Great efforts have been made to integrate the device to the existing ecosystem. For example, new programming model with special CPU instructions (e.g., `clwb`, `clflush`, and `clflushopt`) were introduced to provide durability guarantees. In addition, new persistence domains were defined to better understand the persistence guarantees offered by these devices. Figure 1(a) shows the two persistence domain supported on Intel platforms. **Asynchronous DRAM Refresh (ADR)** domain specifies that all updates that reach the write pending queue in the integrated memory controller are guaranteed to be persisted in an event of system failure, whereas **Enhanced ADR (eADR)** domain specifies that all updates in the caches and memory controller are persisted in an event of system failure. Therefore, there is no need for developers to explicitly flush cachelines using special CPU instructions. Note that eADR is a relatively recent feature that may not be available in all platforms. Also, while eADR may provide stronger persistence guarantee at the hardware level, PM software still needs to be carefully designed to achieve desired high level properties and correctness guarantees (e.g., atomicity of a transaction) [34]. These efforts have inspired the development of many new PM-optimized software [19, 61].

Most recently, vendor-neutral interfaces such as CXL have been pushing the evolution of PM devices further. For example, CXL 2.0 specification has included the support for PM devices [4, 91]. These CXL-based PM devices are expected to be compatible with the existing NVDIMM specifications, and they will still show up as special memory devices to the OS kernel. Therefore,

although Intel is winding down the Optane DCPMM business with limited support in the next 3 to 5 years [29], the community is expecting to see CXL-based PM devices in the near future that are compatible with the NVDIMM ecosystem [4, 29, 34, 91].

2.3 PM Software Abstractions

To support PM devices in the storage system, multiple subsystems in the Linux kernel were modified. New Unified Extensible Firmware Interface specifications were introduced, and a new NVDIMM Firmware Interface Table driver was added to the OS kernel [16]. Moreover, to manage and access PM devices efficiently, the Linux kernel exposes the PM storage space over multiple abstractions. Specifically, the PM devices attached to NVDIMM can be configured into either interleaved or non-interleaved mode. In the interleaved mode (Figure 1(b)), an NVDIMM region is created over multiple NVDIMM devices; in the non-interleaved mode, each device has one NVDIMM region. Additionally, multiple namespaces may be created within one NVDIMM region, which is conceptually similar to creating multiple partitions on one block device.

The NVDIMM devices are registered as special memory devices in the Linux kernel. Each NVDIMM namespace maintains a dedicated memory area to store **Page Frame Number (PFN)** metadata, which supports the necessary address translation when accessing the storage media. Moreover, to bypass the page cache and enable direct access to PM, major file systems such as Ext4 and XFS have introduced the direct-access (DAX) feature based on the traditional Direct IO feature. As will be shown in our study, however, it is challenging to implement such PM-oriented optimizations correctly due to the system complexity.

3 METHODOLOGY

In this section, we describe how we collect the dataset for study (Section 3.1), how we characterize the PM-related patches and bugs (Section 3.2), and the limitations of the methodology (Section 3.3).

3.1 Dataset Collection and Refinement

All changes to the Linux kernel occur in the form of patches [23], including but not limited to bug fixes. We collect PM-related patches from the Linux source tree for study via four steps as follows:

First, we collected all patches committed to the Linux kernel between January 2011 and December 2021, which generated a dataset containing about 772,000 patches.

Second, to effectively identify PM-related patches, we refine the dataset using a wide set of PM-related keywords, such as "`persistent memory`," "`pmem`," "`dax`," "`ndctl`," "`nvdimm`," "`clflushopt`," and so on. The resulting dataset contains 3,050 patches. Note that this step is similar to the keyword search in previous studies [55, 80].

Third, to prune the potential noise, we refine the dataset further by manual examination. Each patch is analyzed at least twice by different researchers, and those irrelevant to PM are excluded based on our domain knowledge. The final dataset contains 1,553 PM-related patches in total.

3.2 Dataset Analysis and Experiments

Based on the 1,553 PM-related patches, we conduct a comprehensive study to answer four sets of questions as follows:

- Overall Characteristics: What are the purposes of the PM-related patches? How many of them are merged to fix correctness issues (i.e., PM bugs)?
- Bug Characteristics: What types of PM-related bugs existed in the Linux kernel? What are the bug patterns and consequences? How are they fixed?

Table 1. Three Categories of PM-related Patches and the Percentages

Category	Description	Overall
Bug	Fix existing correctness issues (e.g., misalignment of PM regions, race on PM pages)	352 (22.7%)
Feature	Add new features or improve efficiency of existing ones (e.g., extend device flags, reduce write overhead)	597 (38.4%)
Maintenance	Polish source code, compilation scripts, and documentation (e.g., delete obsolete code, fix compilation errors)	604 (38.9%)
	Total	1553

- Reproducibility: Can the PM-related bugs be reproduced for future research? What are the critical conditions for manifesting the issues?
- Implications: What are the limitations of existing PM bug detection tools? What are the opportunities?

To answer these questions, we manually analyzed each patch in depth to understand its purpose and logic. The patches typically follow a standard format containing a description and code changes [23], which enables us to characterize them along multiple dimensions. For patches that contain limited information, we further looked into relevant source code and design documents. Moreover, we conduct experiments to validate the reproducibility of PM bugs and the capability of state-of-the-art bug detectors. We present our findings for the four sets of questions above in Section 4, Section 5, Section 6, and Section 7, respectively.

3.3 Limitations

The results of our study should be interpreted with the method in mind. The dataset was refined via PM-related keywords and manual examination, which might be incomplete. Also, we only studied PM bugs that have been triggered and fixed in the mainline Linux kernel, which is biased: There might be other latent (potentially trickier) bugs not yet discovered. Nevertheless, we believe our study is one important step toward addressing the challenge. We release our results publicly to facilitate follow-up research [3].

4 PM PATCH OVERVIEW

We classify all PM-related patches into three categories as shown in Table 1: (1) "`Bug`" means fixing existing correctness issues (e.g., misalignment of NVDIMM namespaces); (2) "`Feature`" means adding new features (e.g., extend device flags) or improving the efficiency of existing designs; (3) "`Maintenance`" means code refactoring, compilation, or documentation update, and so on.

Overall, the largest category in our dataset is "`Maintenance`" (38.9%), which is consistent with previous studies on Linux file system patches [78]. This reflects the significant effort needed to keep PM-related kernel components well maintained. We find that the majority of maintenance patches are related to code refactoring (e.g., removing deprecated functions or driver entry points), and occasionally the refactoring code may introduce new correctness issues that need to be fixed via other patches.

The second largest category is "`Feature`" (38.4%). This reflects the significant changes needed to add PM to the Linux ecosystem that has been optimized for non-PM devices for decades. One interesting observation is that many (40+) feature patches are proactive (e.g., *"In preparation for adding more flags, convert the existing flag to a bit-flag"* [5]), which may imply that PM-based extensions tend to be well planned in advance. Also, most recent feature patches are related to

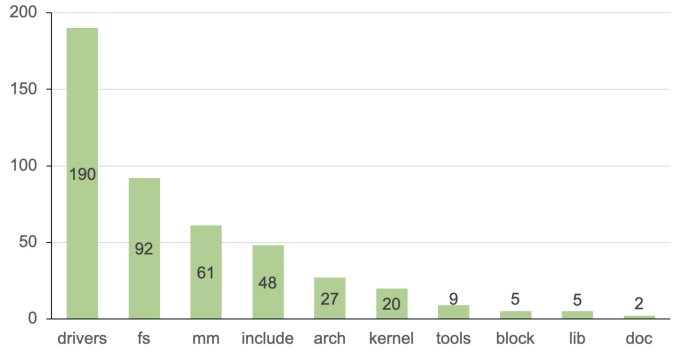


Fig. 2. Counts of PM bug patches in the kernel source tree. The figure shows that PM bug patches involve many major kernel subsystems, with drivers and file systems contributing the most.

supporting PM devices on the CXL interface [4], which indicates the rapid evolution of PM relevant techniques.

The "Bug" patches, which directly represent confirmed and resolved correctness issues in the kernel, account for a non-negligible portion (i.e., 22.7% overall). We analyze this important set of patches further in the next section.

5 PM BUG CHARACTERISTICS

5.1 Where Are the Bugs

Figure 2 shows the distribution of PM bug patches in the Linux kernel source tree. For clarity, we only show the major top-level directories in Linux, which represent major subsystems (e.g., "`fs`" for file systems and "`mm`" for memory management). In case a patch modifies multiple files across different directories (which is not uncommon as will be discussed in Section 5.4.1), we count it toward all directories involved. Therefore, the total count is larger than the number of PM bug patches.

We can see that "`driver`" is involved in most patches, which is consistent with previous studies [43]. In the PM context, this is largely due to the complexity of adding the `nvdimm` driver and the support for the CXL interface. Also, "`fs`" accounts for the second most patches, largely due to the complexity of adding DAX support for file systems [6]. The fact that PM bug patches involve many major kernel subsystems implies that we cannot only focus on one (e.g., "`fs`") to address the challenge.

We also count the occurrences of individual files involved in the bug patches. Table 2 shows the top 10 most "buggy" files based on the occurrences and the average **lines of code (LoC)** changed per 100 LoC, which verifies that adding DAX and NVDIMM support are the two major sources of introducing PM bugs in the kernel. In addition, the table also shows a fine-grained view (i.e., individual files) on where the bugs exist within a subsystem. Therefore, applying bug detection techniques (e.g., static code analysis) on these specific files and/or code paths may help identify most issues.

5.2 Bug Pattern

To build reliable and secure PM systems, it is important to understand the types of bugs that occurred. We analyze the PM bug patches in depth and classify the bugs into five main types (Table 3) as follows: *Hardware Dependent*, *Semantic*, *Concurrency*, *Memory*, and *Error Code*. Each type includes multiple subtypes. The last column of Table 3 shows the major subsystems affected

Table 2. Top 10 Most "Buggy" Files

File Name	# of Occurrence	LoC Changed per 100 LoC
fs/dax.c	41	5.08
drivers/nvdimm/pfn_devs.c	22	2.62
drivers/nvdimm/bus.c	21	1.8
drivers/nvdimm/pmem.c	18	3.23
drivers/acpi/nfit/core.c	16	0.7
drivers/nvdimm/region_devs.c	15	1.95
drivers/nvdimm/namespace_devs.c	15	0.8
drivers/dax/super.c	14	3.27
mm/memory.c	13	0.53
drivers/nvdimm/btt.c	12	2.44

The table shows that 9 of the 10 files containing the most PM bugs are related to the "dax" or "nvdimm" supports in the kernel.

Table 3. Classification of PM Bug Patterns

Type	Subtype	Description	Major Subsystems
Hardware Dependent	Specification	misunderstand specification (e.g.: ambiguous ACPI specifications)	drivers, arch, include
	Alignment	mismatch b/w abstractions of PM device (e.g.: misaligned NVDIMM namespace)	drivers, mm, arch
	Compatibility	Device or architecture compatibility issue	drivers, arch, mm
	Cache	Misuse of cache related operations (e.g.: miss cacheline flush)	arch, mm, drivers
Semantic	Logic	improper design (e.g.: wrong design for DAX PMD mgmt.)	drivers, fs, mm
	State	incorrect update to PM State	fs, mm
	Others	other semantic issues (e.g.: wrong function / variable names)	drivers, fs
Concurrency	Race	data race issues involving DAX IO	fs, mm, drivers
	Deadlock	deadlock on accessing PM resource	drivers, mm, fs
	Atomicity	violation of atomic property for PM access	drivers, fs, mm
	Wrong Lock	use wrong lock for PM access	fs, drivers, block
	Order	violation of order of multiple PM accesses	fs
	Double Unlock	unlock twice for PM resource	drivers
	Miss Unlock	forget to unlock PM resource	drivers
Memory	Null Pointer	dereference null PM / DRAM pointer	drivers, fs, mm
	Resource Leak	PM / DRAM resource not released	drivers, mm, arch
	Uninit. Read	read uninitialized PM / DRAM variables	drivers, fs
	Overflow	overrun the boundary of PM/DRAM struct.	drivers, fs, include
Error Code	Error Return	no / wrong error code returned	drivers, fs, kernel
	Error Check	miss / wrong error check	drivers, fs, mm

The last column shows the major subsystems (up to 3) affected by the bugs.

by each type of bugs. For clarity, the column only lists up to three subsystems for each type. We can see that the same type of bugs may affect multiple subsystems (e.g., "`drivers`," "`fs`," "`mm`"), and each subsystem may suffer from multiple types of bugs.

Figure 3 shows the relative percentages of the five main types. We can see that the *Semantic* type is dominant (49.7%), followed by the *Hardware Dependent* type (19.0%). Similarly, Figure 4 further

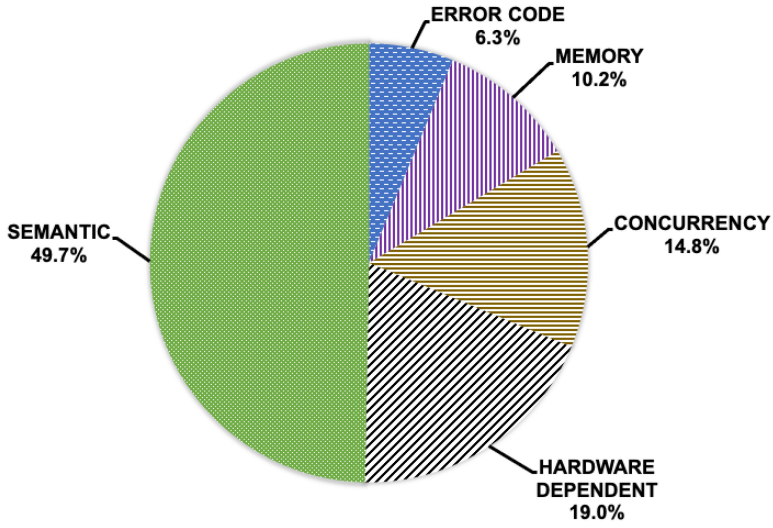


Fig. 3. Percentages of PM bug types.

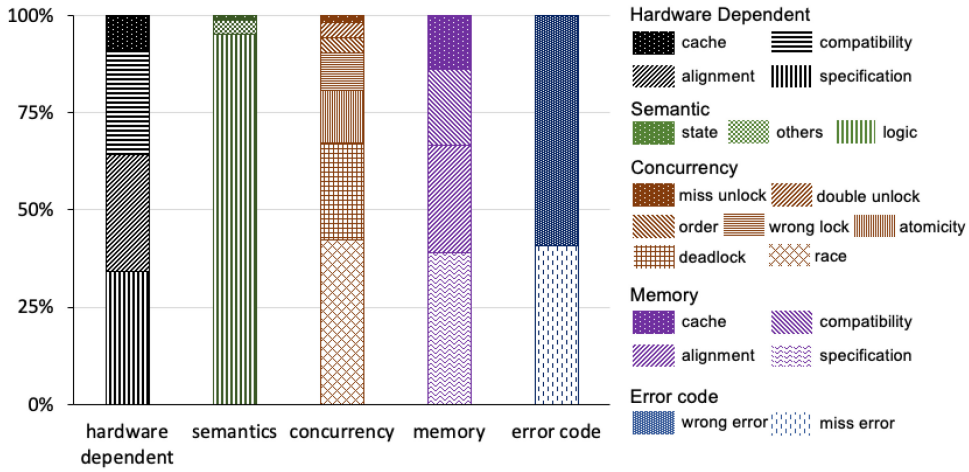


Fig. 4. Percentages of PM bug subtypes. The five bars represent the five main types, each of which consists of multiple subtypes.

shows the relative percentages of the subtypes within each main type. We elaborate on multiple representative types below based on these classifications, with an emphasis on the ones that are different from previous studies [78, 80].

5.2.1 Hardware Dependent. Compared to previous studies [78, 80], the most unique pattern observed in our dataset is *Hardware Dependent*, which accounts for 19.0% of PM bugs (Figure 3). There are four subtypes, including *Specification* (34.3% of *Hardware Dependent*), *Alignment* (29.9%), *Compatibility* (26.9%), and *Cache* (8.9%), which reflects four different aspects of challenge for integrating PM devices correctly to the Linux kernel.

drivers/nvdimm/bus.c

```
1    u32 nd_cmd_out_size(...) {
2  -       if(out_field[1] - 8 == remainder) {
3  +       if(out_field[1] - 4 == remainder) {
4               return remainder;
5  -       return out_field[1] - 4;
6  +       return out_field[1] - 8;
```

(a) **A Specification Bug Example**

drivers/nvdimm/pfn_devs.c

```
1    nd_pfn_validate(...) {
2  +     unsigned long start_pad = le32_to_cpu(pfn_sb->start_pad);
3        ...
4  -     if((align && !IS_ALLOWED(offset, align))...)
5  +     if((align && IS_ALIGNED(nsio->res.start + offset
6  +                                     + start_pad, align))...)
```

(b) **An Alignment Bug Example**

arch/x86/lib/usercopy_64.c

```
1    long __copy_user_flushcache(...) {
2        if(size < 8) {
3            if(!IS_ALIGNED(dest, 4) || size != 4) {
4  -             clean_cache_range(dst, 1);
5  +             clean_cache_range(dst, size);
6            } else {
```

(c) **A Cache Bug Example**

Fig. 5. Three examples of hardware-dependent bugs. (a) Caused by the ambiguity of the ACPI specification; (b) caused by misalignment between the PFN abstraction and the NVDIMM namespace; (c) caused by inaccurate flushing of cachelines.

Specification is the largest subtype of *Hardware Dependent* bugs (34.3%). Figure 5(a) shows an example caused by the ambiguity of PM hardware specification. In this case, the PM device uses **Address Range Scrubbing (ARS)** [1] to communicate errors to the kernel. ACPI 6.1 specification [1] requires defining the size of the output buffer, but it is ambiguous if the size should include the 4-byte ARS status or not. As a result, when the `nvdimm` driver should have been checking for "`out_field[1] - 4`," it was using "`out_field[1] - 8`" instead, which may lead to a crash.

In terms of the other three subtypes, we find that *Alignment* issues are typically caused by the inconsistency between various abstractions of PM devices (e.g., PM regions, namespaces). Figure 5(b) shows an issue due to a misalignment between the PFN device abstraction and the NVDIMM namespace. While validating the PFN device alignment, the method does not take into account of the padding at the beginning of the namespace. This inconsistency makes the NVDIMM device inaccessible.

Compatibility issues often arise when the DAX functionality conflicts with the underlying CPU architecture or PM device. For example, the DAX support requires modifying page protection bits in the **page table entries (PTEs)**, which depends on CPU architectures. However, PowerPC architecture does not allow for such modification to valid PTEs. A check within the memory management subsystem triggers a kernel warning in this scenario. This bug was fixed by invoking a PowerPC-specific routine when modifying PTEs.

Cache issues are caused by misuse of cache-related operations (e.g., `clflushopt`). The cache-related operations have been the major focus of existing studies on user-level PM software [69, 88]. Nevertheless, we find that the *Cache* subtype only accounts for 8.9% of *Hardware Dependent* bugs in our dataset (Figure 4). Moreover, we find that the *Cache* bug pattern is different from the typical

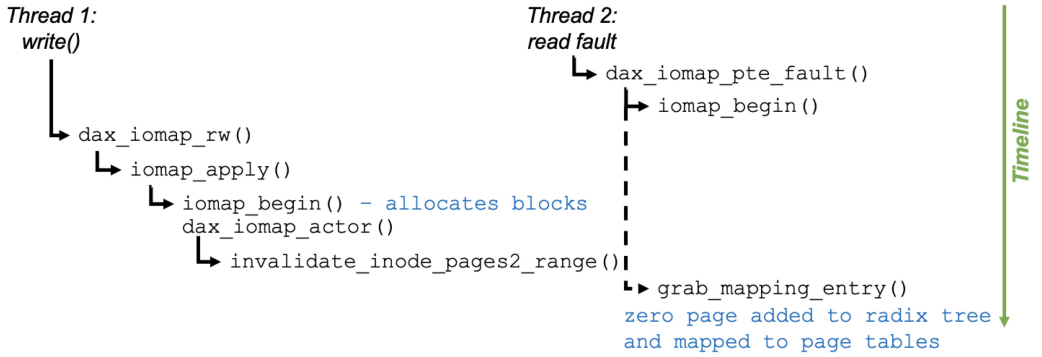


Fig. 6. A concurrency bug example in the PM context. This bug was caused by a race condition involving DAX IO operations: Thread 1 attempts to write data to a given address range, while Thread 2 attempts to read data from the same address range.

pattern widely studied in the literature. Existing studies mostly focus on the cases where a specific operation (cacheline flush or memory fence) is missing. In our dataset, however, the issues may arise due to partially flushing data from the volatile CPU cache to PM media. Figure 5(c) shows an example. In this example, the function `clean_cache_range` in line 4 is expected to flush the cacheline holding the data of variable `dst`. However, if the data occupy two cachelines, then the code in line 4 only flushes data in the first cacheline. A crash at this point may lead to incomplete data on persistent media. This bug was fixed (line 5) by providing the size of data as input to `clean_cache_range`. Such subtle granularity issues will likely require additional innovations on bug detection to handle.

5.2.2 Other Types. In addition to *Hardware Dependent*, we find that PM bugs may follow the classic *Semantic*, *Concurrency*, *Memory*, and *Error Code* patterns observations in traditional file systems and/or user-level applications [78, 80], but the root causes may be different due to the different contexts. For example, *Concurrency* bugs in our dataset are specific to the PM environment. In particular, we find that the majority of *Concurrency* PM bugs are caused by race conditions between the DAX page handler and regular IO operations. Figure 6 shows one specific example involving two threads. In this case, Thread 1 invokes a write syscall that allocates blocks on PM, and Thread 2 invokes a read to the same PM blocks that triggers a page fault. When Thread 1 is updating the block mappings, Thread 2 should wait until the update completes. However, due to the lack of proper locking, Thread 2 instead maps hole pages to the page table, which results in reading zeros instead of data written in place. This may cause crash-consistency issues. The bug was fixed by locking the exception entry before mapping the blocks for a page fault. In this way, either the writer will be blocked until read finishes or the reader will see the correct PM blocks updated by the writer.

Figure 7 shows three additional examples of classic bug patterns in the PM context. Specifically, Figure 7(a) shows a *Memory Overflow* bug where an offset was missed when calculating a PM address in the NVDIMM driver, and this miscalculation led to an out-of-bound access error, resulting in a kernel crash. In Figure 7(b) (*Error Code* bug), the return code was not captured when the struct `nd_class` is invalid, which affected the error handling later (i.e., no error code returned). In Figure 7(c) (*Semantic* bug), a wrong size is used for the request for accessing the NVDIMM block device, which limited the max I/O size to PM to 1,024 sectors resulting in an inconsistent read or write. Overall, these classic bug patterns suggest that "history repeats itself," and more efforts are needed to address the classic issues in the PM context.

drivers/nvdimm/pmem.c

```
1   static void write_pmem(...)
2     while (len) {
3 -     chunk = min_t(len,PAGE_SIZE);
4 +     chunk = min_t(len,PAGE_SIZE - off);
5       ...
6 -     pmem_addr += PAGE_SIZE;
7 +     pmem_addr += chunk;
```

(a) **A Memory Overflow Bug Example**

drivers/nvdimm/bus.c

```
1   int __init nvdimm_bus_init(...) {
2     nd_class = class_create(...);
3 -   if(IS_ERR(nd_class))
4 +   if(IS_ERR(nd_class)) {
5 +     rc = PTR_ERR(nd_class);
6       goto err_class;
7 +   }
```

(b) **An Error Code Bug Example**

drivers/nvdimm/pmem.c

```
1   int pmem_attach_disk(...) {
2       ...
3       blk_queue_make_request(pmem->queue, pmem_make_request);
4 -     blk_queue_max_hw_sectors(pmem->queue, 1024);
5 +     blk_queue_max_hw_sectors(pmem->queue, UINT_MAX);
6       disk = alloc_disc(0);
```

(c) **A Semantic Bug Example**

Fig. 7. Three different types of classic bug patterns in the PM context. (a) Memory Overflow: An offset was missed when calculating a PM address in the NVDIMM driver, which caused an out-of-bound access error. (b) Error Code: The return code is not captured when the struct `nd_class` is invalid, which affected the error handling. (c) Semantic: The max size of an I/O request for the PM device is wrong.

5.3 Bug Consequence

To understand how severe the PM bugs are, we classify them based on the symptoms reported in the patches. We find that there are eight types of consequence, including *Missing Device*, *Inaccessible Device*, *Security*, *Corruption*, *Crash*, *Hang*, *Wrong Return Value*, and *Resource Leak*. We elaborate on the first four types as they are relatively more unique to our dataset, while the others are similar to previous studies [78].

First, *Missing Device* implies the kernel is unable to detect PM devices, which is often the consequence of hardware dependent bugs. For example, the `e820_pmem` driver is responsible for registering resources that surface as `pmem` ranges. However, the buggy "`e820_pmem_probe`" method may fail to register the `pmem` ranges into the **System-Physical Address (SPA)** space, which makes the PM device not recognizable by the kernel.

Second, *Inaccessible Device* means the PM device is detectable by the kernel but not accessible. For example, the "`start_pad`" variable was introduced in "`struct nd_pfn_sb`" of the `nvdimm` driver to record the padding size for aligning namespaces with the Linux memory hotplug section. But the buggy "`nd_pfn_validate`" method of the driver does not check for the variable, which leads to an alignment issue and makes the namespace not recognizable by the kernel.

Third, in terms of *Security*, we observe two interesting issues. In one case, write operations may be allowed on read-only DAX mappings, which exposes wrong access permissions to the end user. In another case, an NVDIMM's security attribute remains in "unlocked" state even when the admin issues an "overwrite" operation, which could potentially allow malicious accesses to the PM device.

Fourth, *Corruption* means the data or metadata stored on PM devices are corrupted, which could lead to permanent data loss if there is no additional backup available on other systems. One special type of corruption is the *Crash-Consistency* issues, which means tricky corruptions (i.e., data or metadata inconsistencies) triggered by a crash event (e.g., power outage, kernel panic). Such issues have been investigated intensively in the literature due to its importance [85, 111, 113], and we

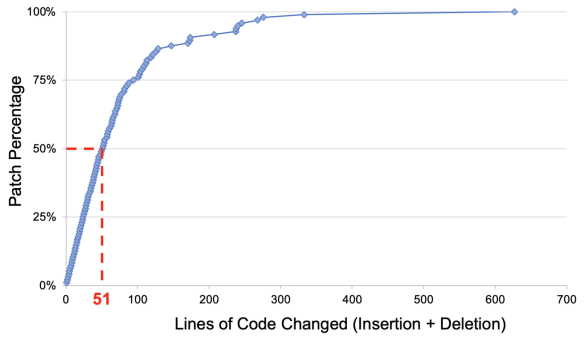


Fig. 8. Size distribution of PM bug patches.

Table 4. Scope of PM Bug Patches

File Count	1	2	3	4	5	> 5
Patch %	66.0%	13.2%	8.3%	6.0%	2.9%	3.6%
Dir. Count	1	2	3	4	5	> 5
Patch %	79.2%	12.7%	5.7%	1.6%	0.5%	0.3%

This table shows the % of bug patch involving different counts of files or directories.

have observed multiple crash-consistency issues in our dataset as well. For example, the routine `dax_mapping_entry_mkclean` may fail to clean and write protect DAX PMD entries; consequently, memory-mapped writes to DAX PMD pages may not be flushed to PM even when a sync operation is invoked. As a result, a crash may leave the system in an inconsistent state.

5.4 Bug Fix

5.4.1 How Difficult It Is to Fix PM Bugs. Measuring the complexity of fixing PM bugs is challenging as it requires deep domain knowledge and may depend on the developers' capability and other constraints (e.g., priority of tasks). Inspired by existing studies [43, 73, 78], we calculate three simple but quantitative metrics as follows, which might reflect the complexity to some extent.

Bug Patch Size. We define the patch size as the sum of lines of insertion and deletion in the patch. Figure 8 shows the distribution of bug patch sizes. We can see that most bug patches are relatively small. For example, 50% patches have less than 51 lines of insertion and deletion code (LoC). However, compared to traditional non-PM file system bug patches where 50% are less than 10 LoC [78], the majority of PM bug patches tend to be larger.

Bug Patch Scope. We define the patch scope as the counts of files or directories involved in the patch. For simplicity, we only count the top-level directories in the Linux source tree. Table 4 shows the patch scopes. We can see that most patches only modified one file (66.0%) or files within one directory (79.2%). However, 3.6% patches may involve more than five files. Moreover, a non-negligible portion of patches involve more than one directories (20.8%). Since different directories represent different kernel subsystems, this implies that fixing these PM bugs are non-trivial. For comparison, we randomly sample 100 GPU-related bug patches and measure their scope, too. We observe that only 5% of the sampled GPU patches involve more than one directory, which is much less than the 20.8% cross-subsystem PM bug patches.

drivers/nvdimm/pfn_devs.c

```
1   int nd_pfn_validate(...) {
2       ...
3 -     if(!is_power_of_2(offset)
4 -             || offset < PAGE_SIZE) {
5 +     if((nd_pfn->align
6 +             && !IS_ALIGNED(offset, nd_pfn->align))
7 +             || !IS_ALIGNED(offset, PAGE_SIZE)) {
8 +                 dev_err(...)
9                   return -ENXIO;
```

(a) **An Alignment Bug Fix**

fs/xfs/xfs_aops.c

```
1   int __xfs_get_blocks (...) {
2       ...
3       if ( ...
4           (imap.startblock == HOLESTARTBLOCK ||
5 -          imap.startblock == DELAYSTARTBLOCK))) {
6 +          imap.startblock == DELAYSTARTBLOCK) ||
7 +          (IS_DAX(inode) && ISUNWRITTEN(&imap)))) {
8               err = xfs_iomap_write_direct(...)
```

(b) **A Data Race Bug Fix**

Fig. 9. Two examples of refining sanity checks to fix PM bugs: (a) Alignment bug fix and (b) data race bug fix.

Although many PM bug patches in our dataset involves changes in multiple kernel subsystems, this does not imply that only PM bug patches can involve multiple kernel subsystems. It is possible that there are complicated non-PM bugs that may also require modifications across multiple subsystems to fix. We leave further comparison of PM bugs and non-PM bugs in terms of the patch scope as future work.

Time-to-Fix. Most patches in our dataset do not contain the information *when* the bug was first discovered. However, we find that 48 bug patches include links to the original bug reports, which enables us to measure the time-to-fix metric. We find that PM bugs may take 6 to 48 days to fix with an average of 24 days, which further implies the complexity. There are other sources that may provide more complete time-to-fix information (e.g., Bugzilla [12]), which we leave as future work. Note that similar to measuring "how long do bugs live" in the classic study of Linux and OpenBSD kernel bugs [43], human factors such as the developer's availability and capability may affect the time-to-fix metric; therefore, there is unlikely a linear relationship between the time-to-fix metric and the bug complexity. In other words, how to measure the complexity of bug precisely is still an open challenge. However, we find that during the time to fix window, there were often in-depth discussions among developers, which requires substantial domain knowledge and trial-and-error steps; therefore, we believe that the long time-to-fix value still reflects the need of better debugging support to certain extent, similar to the observations from previous studies [43, 73].

5.4.2 Fix Strategy. We find that the strategies for fixing PM bugs often vary a lot depending on the specific bug types (Table 3). However, we also observe that different types of PM bugs may be fixed by one common strategy: refining sanity checks. For example, Figure 9(a) shows an *Alignment* bug fix. In this example, a PM device was mistakenly disabled due to an ineffective sanity check (`is_power_of_2`). The bug was fixed by replacing the original sanity check with an accurate one (`IS_ALIGNED`). Similarly, Figure 9(b) shows a data race bug that was triggered when there were concurrent write faults to same DAX pages. The error occurs, because the DAX page fault was not aware of pre-allocated (unwritten data) and just-allocated blocks. Therefore, a data race may end up zeroing the pages, similarly to the case described in Figure 6 (Section 5.2.2). The bug was fixed in the filesystem layer by refining the sanity check to identify unwritten DAX pages. Similar fixes have been applied to other bugs triggered by check violations.

We also find that developers may *temporarily* "fix" a PM bug by disabling a PM feature. For example, to avoid a race condition in handling **transparent huge pages (THP)** over DAX, developers make the THP support over DAX dependent on `CONFIG_BROKEN`, which means if `CONFIG_BROKEN` is disabled (common case), then the feature is disabled, too. The developers even mention that a major re-work is required in the future, which implies the complexity of actually fixing the bug.

Table 5. Five Metrics for Selecting PM Bug Candidates for Reproducibility Experiments

Metric ID	Description	Bug #
1Config	Configuration parameters of PM device & relevant software	20
2HW	Information of specific PM hardware type (e.g., NVDIMM-N)	12
3Emu	Information of PM emulation platform (e.g., QEMU)	4
4Test	Information of test suite and/or test case used (e.g., xfstests)	18
5Step	Information of necessary steps to reproduce the bug	18
	Total Unique	39

6 PM BUG REPRODUCIBILITY

To better understand the conditions for manifesting the issues and help derive effective remedy solutions, we further perform a set of reproducibility experiments. We identify a subset of bug patches with relatively complete information and attempt to reproduce them experimentally. We find that configuration parameters in different storage utilities (e.g., `mkfs` for creating a file system, `ndctl` for managing the `libnvdimm` subsystem) are critically important for manifesting the issues, which suggests that it is necessary to take into account the configuration states when developing relevant testing or debugging tools.

As shown in Table 5, we look for five pieces of information from the bug patches when selecting candidates for the reproducibility experiments, which includes configuration parameters (*1Config*), hardware information (*2HW*), emulation platform (*3Emu*), test suite information (*4Test*), and reproducing steps information (*5Step*). Based on the five metrics, we identify 39 bug candidates that contain relatively complete information for the experiments.

Table 6 summarizes the reproducibility results. At the time of this writing, we are able to reproduce 11 of the 39 candidates based on the information derived from the patches (✓in the last column). Most interestingly, we find that the configuration parameters of storage utilities (i.e., `mkfs`, `mount`, `ndctl`) and workloads are critically important for triggering the bugs, which are summarized in the four columns of "Critical Configurations."

For example, in bug #15, when the file system is mounted with both DAX and read-only parameters, accessing the file system results in a "Segmentation fault," because the DAX fault handler attempts to write to the journal in read-only mode. As another example, the `ndctl` utility enables configuring the `libnvdimm` subsystem of the Linux kernel. In bug #35, a resource leak was observed when an existing NVDIMM namespace was reconfigured to "device-dax" mode via `ndctl`.

Note that Table 6 only shows the necessary configurations for triggering the bug cases, which may not be sufficient. In fact, there are many other factors that can make a case un-reproducible in our experiments. For example, we are unable to reproduce cases that require test cases from the `ndctl` test suite [17]. These test cases require "nfit_test.ko" kernel module, which is not available in the generic Linux kernel source tree. Also, some cases are dependent on specific architectures (e.g., PowerPC) that are incompatible with our experimental systems. In addition, many concurrency bugs such as data races cannot be easily reproduced due to the nondeterminism of thread interleavings.

Overall, we can see that most of the bug cases in Table 6 require specific configurations to trigger, which suggests the importance of considering configurations for testing and debugging [40, 84, 92, 100]. We summarize all the reproducible bug cases, including the necessary triggering conditions and scripts, in a public dataset to facilitate follow-up research [3].

Table 6. Results of Reproducing 39 PM Bug Cases

Bug ID	Bug Type	Critical Configurations				R?
		mkfs	mount	ndctl	workload	
1	Semantic - State	—	—	—	`mmap(MAP_SHARED)`	✗
2	Concurrency - Wrong lock	—	—	—	—	✗
3	Error code - Error return	—	—	—	—	✗
4	Semantic - Logic	—	`-o dax`	—	—	✓
5	Semantic - Logic	—	`-o dax`	—	—	✓
6	Semantic - Logic	—	—	—	—	✗
7	Hardware - Alignment	—	—	`-a 4096`	—	✗
8	Semantic - Logic	—	—	—	`fallocate (FALLOC_FL_ZERO_RANGE)`	✗
9	Hardware - Specification	—	—	—	—	✗
10	Hardware - Cache	—	`-o dax`	—	—	✗
11	Hardware - Alignment	—	—	`-m devdax -a 4K`	—	✗
12	Semantic - Logic	—	—	`-m devdax -a 4K`	—	✗
13	Concurrency - Race	—	`-o dax`	—	—	✗
14	Semantic - Logic	—	`-o dax`	—	—	✓
15	Semantic - Logic	—	`-o dax,ro`	—	—	✓
16	Concurrency - Deadlock	—	`-o dax`	—	—	✗
17	Semantic - Logic	`-O inline_data`	`-o dax`	—	—	✓
18	Semantic - Logic	—	`-o dax, nodelalloc`	—	—	✓
19	Error code - Error check	—	`-o dax`	—	`open(O_TRUNC)`	✗
20	Hardware - Alignment	—	—	`-m devdax -a 1G`	—	✗
21	Semantic - Logic	`-t xfs`	`-o dax`	`-l 4K`	—	✗
22	Semantic - Logic	—	—	—	`blockdev --setro`	✓
23	Hardware - Specification	—	—	—	—	✗
24	Semantic - Logic	—	—	—	—	✗
25	Concurrency - Race	`-t xfs`	`-o dax`	—	—	✗
26	Concurrency - Race	—	—	—	—	✗
27	Semantic - Logic	—	—	`-s < 16MB`	—	✗
28	Semantic - Logic	—	—	`-e -m devdax`	—	✗
29	Concurrency - Atomicity	—	—	`create-namespace destroy-namespace`	—	✓
30	Concurrency - Deadlock	—	—	`create-namespace destroy-namespace`	—	✓
31	Semantic - Logic	—	`-o dax`	—	`mmap(MAP_PRIVATE)`	✗
32	Semantic - Logic	—	`-o dax`	—	`mmap(MAP_SYNC)`	✗
33	Hardware - Compatibility	—	—	`-m devdax -a 16M`	—	✗
34	Error code - Error return	—	—	`inject-error`	—	✗
35	Memory - Resource leak	—	—	`-e -m devdax`	—	✓
36	Semantic - Logic	—	—	`sanitize-dimm --overwrite`	—	✗
37	Semantic - Logic	`-O inline_data`	—	—	`chattr +x`	✓
38	Semantic - Logic	—	—	`-e -m devdax`	—	✗
39	Memory - Resource leak	—	—	—	—	✗

The last column *R?* means if the bug case was reproducible (✓) or not (✗). The middle columns show the bug type as well as the critical configurations necessary for triggering the bug cases.

7 IMPLICATIONS ON PM BUG DETECTION

Our study has exposed a variety of PM-related issues, which may help develop effective PM bug detectors and build robust PM systems. For example, since 20.8% PM bug patches involve multiple kernel subsystems, simply focusing on one subsystem is unlikely enough; instead, a full-stack approach is much needed, and identifying the potential dependencies among components would be critical. However, since many bugs in different subsystems may follow similar patterns, capturing one bug pattern may benefit multiple subsystems (see Section 9.1 for more discussions).

As one step to address the PM-related issues identified in the study, we analyze a few state-of-the-art bug detection tools in this section. We discuss the limitations as well as the opportunities for both PM bug detectors and Non-PM bug detectors (Section 7.1 and Section 7.2, respectively). Moreover (Section 7.3), we present our efforts and results on extending one state-of-the-art static bug detector (i.e., Dr. Checker [82]) for analyzing PM drivers, which account for the majority of bug cases in our dataset (Figure 2).

7.1 PM Bug Detectors

Multiple PM-specific bug detection tools have been proposed recently, including PMTest [77], XFDetector [76], and AGAMOTTO [88]. These tools mostly focus on user-level PM programs. We have performed bug detection experiments using these tools, and we are able to verify their effectiveness by reproducing most of the bug detection results reported in the literature. Unfortunately, we find that they are fundamentally limited for capturing the PM bugs in our dataset. For example, XFDetector [76] relies on Intel Pin [21], which can only instrument user-level programs. PMTest [77] can be applied to kernel modules, but it requires manual annotations, which is impractical for major kernel subsystems. AGAMOTTO [88] relies on KLEE [35] to symbolically explore user-level PM programs. While it is possible to integrate KLEE with virtual machines to enable full-stack symbolic execution (as in S2E [42]), novel PM-specific path reduction algorithms are likely needed to avoid the state explosion problem [67]. One recent work Jaaru [53] leverages commit stores, a common coding pattern in user-level PM programs, to reduce the number of execution paths that need to be explored for model checking. Nevertheless, such elegant pattern has not been observed in our dataset due to the complexity of kernel-level PM optimizations (Section 5). Therefore, additional innovations on path reduction are likely needed to apply model checking to detect diverse PM-related bugs in the kernel effectively.

7.2 Non-PM Bug Detectors

Great efforts have been made to detect non-PM bugs in the kernel [13, 25, 27, 63, 66, 85, 99, 101]. For example, CrashMonkey [85] logs the `bio` requests and emulates crashed disk states to test the crash consistency of traditional file systems. As discussed in Section 5.3, such tricky crash consistency issues exist in PM subsystems, too. Nevertheless, extending CrashMonkey to detect PM bugs may require substantial modifications including tracking PM accesses and PM-critical instructions (e.g., mfence), designing PM-specific workloads, among others.

Similarly, fuzzing-based tools have proven to be effective for kernel bug detection [24, 63, 66, 99, 101]. For example, Syzkaller [24] is a kernel fuzzer that executes kernel code paths by randomizing inputs for various system calls and has been the foundation for building other fuzzers; Razzer [63] combines fuzzing with static analysis and detects data races in multiple kernel subsystems (e.g., "`driver`," "`fs`," and "`mm`"), which could potentially be extended to cover a large portion of concurrency PM bugs in our dataset. Since Syzkaller, Razzer, and similar fuzzers heavily rely on virtualized (e.g., QEMU [33]) or simplified (e.g., LKL [15]) environments to achieve high efficiency for kenel fuzzing, one common challenge and opportunity for extending them is to emulate PM devices and interfaces precisely to ensure the fidelity.

Also, Linux kernel developers have incorporated tools such as Kernel Address Sanitizer [25], Undefined Behavior Sanitizer [27], and memory leak detectors (Kmemcheck) [13] within the kernel code to detect various memory bugs (e.g., null pointers, use-after-free, resource leak). These sanitizers instrument the kernel code during compilation and examine bug patterns at runtime. Similarly to other dynamic tools, these tools can only detect issues on the executed code paths. In other words, their effectiveness heavily depends on the quality of the inputs. As discussed in Section 6, many PM issues in our dataset require specific configuration parameters from utilities

```
static const struct file_operations nvdimm_fops = {
        .owner = THIS_MODULE,
        .open = nd_open,
        .unlocked_ioctl = nvdimm_ioctl,
        .compat_ioctl = nvdimm_ioctl,
        .llseek = noop_llseek,
};
```

Fig. 10. Entry points for the libnvdimm driver module.

(e.g., mkfs, ndctl) and workloads (e.g., mmap(MAP_SHARED), open(O_TRUNC)) to trigger, so we believe it is important to consider such PM-critical configurations when leveraging existing kernel sanitizers for detecting the issues exposed in our study.

As discussed above, while various bug detectors have been proposed and used in practice, addressing PM-related issues in our dataset will likely require PM-oriented innovations, including precise PM emulation, PM-specific configuration and workload support, and so on, which we leave as future work.

However, to better understand the feasibility of extending existing tools for PM bug detection, we present our efforts and results on extending one existing bug detector called Dr.Checker [82] in this section. We select Dr.Checker for three main reasons: First, it has proven effective for analyzing kernel-level drivers [82], and, as shown in Figure 2 (Section 5.1), drivers account for the majority of bugs in our dataset. second, Dr.Checker is based on static analysis without dynamic execution, which makes it less sensitive to the limitations discussed in the previous sections (e.g., device emulation, input generation). Third, Dr. Checker employs multiple bug detection algorithms that can detect multiple bug patterns identified in our study (Section 5.2). We name our extension as Dr.Checker+ and release it on Git to facilitate follow-up research on PM bug detection [3].

About Dr.Checker and Its Limitations. Dr.Checker [82] mainly uses two static analysis techniques (i.e., points-to and taint analysis) to detect memory-related bugs (e.g., buffer overflow) in generic Linux drivers. It performs flow-sensitive and context-sensitive code traversal to achieve high precision for driver code analysis. One special requirement for applying Dr.Checker is to identify correct *entry points* (i.e., functions invoking the driver code) as well as the argument types of entry points. For example, Figure 10 shows a VFS interface ("struct file_operations") with function pointers that allow userspace programs to invoke operations on libnvdimm driver module. The functions included in the structure (e.g., "nvdimm_ioctl") are entry points that enable us to manipulate the underlying NVDIMM devices through the driver code. The entry points and their argument types collectively determine the *entry types*, which in turn determines the *taint sources* for initiating relevant analysis. For example, the vanilla Dr. Checker describes an IOCTL *entry type* where the first argument of the entry point function is marked as *PointerArgs* (i.e., the argument points to a kernel location, which contains the tainted data) and the second argument is marked as *TaintedArgs* (i.e., the argument contains tainted data and is referenced directly). This entry type is applicable to entry point functions that have similar signature. In the case of the libnvdimm kernel module, the IOCTL entry type is applicable to two entry point functions (i.e., nd_ioctl, nvdimm_ioctl). In addition, Dr.Checker includes a number of *detectors* to check specific bug patterns based on the taint analysis (e.g., *IntegerOverflowDetector* and *GlobalVariableRaceDetector*).

While Dr.Checker has been applied to analyze a number of Linux device drivers [82], it does not support major PM drivers directly. Table 7 summarizes the seven PM driver modules involved in our study, including nfit.ko, nd_pmem.ko, nd_blk.ko, nd_btt.ko, libnvdimm.ko, device_dax.ko, and dax.ko. These driver modules provide various supports to make PM devices usable on Linux-based systems. For example, dax.ko provides generic support for direct access to

Table 7. List of Major PM Driver Modules Studied in This Work

Driver Module	Description	Ck	Ck+
nfit.ko	Probe NVDIMM devices and register a libnvdimm device tree. Enables libnvdimm driver to pass device specific messages for platform/DIMM configuration.	✗	✓
nd_pmem.ko	Drives a SPA range where memory store operations are persisted. Enables support for DAX feature.	✗	✓
nd_blk.ko	Enables I/O access to DIMM over a set of programmable memory mapped apertures. A set of apertures can access just one DIMM device.	✗	✓
nd_btt.ko	Enables an indirection table that provides power-fail-atomicity of at least one sector (512B). Can be used in front of PMEM or BLK block device drivers.	✗	✓
libnvdimm.ko	Provides generic support for NVDIMM devices, such as discover NVDIMM resources, register and advertise PM namespaces (e.g., /dev/pmemX, /dev/daxX.X, etc.)	✓	✓
device_dax.ko	Support raw access to PM over an mmap capable character device.	✗	✓
dax.ko	Provides generic support for direct access to PM.	✗	✓

The last two columns show whether the module can be supported (✓) or not (✗) by the vanilla Dr.Checker (**Ck**) and our extended version Dr.Checker+ (**Ck+**). The vanilla Dr.Checker can only support the `ioctl` entry type used in `libnvdimm.ko`.

PM, which is critical for implementing the DAX feature for Linux file systems including Ext4 and XFS. As discussed in Section 5 and Section 6, these PM drivers tend to contain the most PM bugs in our dataset, and many PM bugs require PM driver features to trigger (e.g., `-o dax`). Based on our study of the bug patterns and relevant source code, we find that these PM drivers have much more diverse entry types compared to the embedded drivers analyzed by the vanilla Dr.Checker [82]. As shown in the second to the last column of Table 7, the vanilla Dr.Checker [82] only supports one entry type (i.e., `ioctl(file))` used in the `libnvdimm` module, leaving the majority of PM driver code unattended.

Extending Dr.Checker to Dr.Checker+. To make Dr.Checker works for the major PM drivers, we manually examine the source code of PM drivers and identify critical entry points. As summarized in Table 8, we have added five new entry types (i.e., DEV_OP, DAX_OP, BLK_OP, GETGEO, MOUNT) besides the original IOCTL entry type. The new entry types include the major entry point functions used in the PM driver modules. Moreover, we identify the critical input arguments and map them to the appropriate taint types defined by Dr.Checker (i.e., *TaintedArgs* for arguments directly passed by the userspace, and *PointerArgs* and *TaintedArgsData* for arguments pointing to a kernel memory location that contains tainted data). In addition, to make Dr.Checker work for newer versions of Linux kernel, we have ported the implementation to the latest version of the LLVM/Clang compiler infrastructure [26]. Overall, as summarized in the last column of Table 7, the enhanced Dr. Checker+ is able to support bug detection in the seven major PM driver modules based on our comprehensive study of the PM driver bugs and the relevant source code.

7.3 Extending Dr. Checker for Analyzing PM Kernel Modules

Experimental Results of Dr.Checker+. We have applied the extended Dr.Checker+ to analyze seven major PM kernel modules in Linux kernel v5.2. In this set of experiments, we applied four detectors: (1) *IntegerOverflowDetector* checks for tainted data used in operations (e.g., `add`, `sub`, or `mul`) that may cause an integer overflow or underflow, (2) *TaintedPointerDereferenceChecker* detects pointers that are tainted and directly dereferenced, (3) *GlobalVariableRaceDetector* checks for global variables that are accessed without a mutex, and (4) *TaintedSizeDetector* checks for tainted data that is used as a size argument in any of the `copy_to_` or `copy_from_` functions that may

Table 8. Entry Types Identified to Support Dr.Checker for Analyzing PM Drivers

Entry Type	Kernel Module	Entry Point Function(s)	Tainted Input Argument (s)	Taint Type
DEV_OP*	nfit.ko	acpi_nfit_add, acpi_nfit_probe acpi_nfit_remove	struct acpi_device *adev	PointerArg
	nd_blk.ko	nd_blk_probe, nd_blk_remove	struct device *dev	PointerArg
	libnvdimm.ko	nvdimm_release, nvdimm_probe		
	device_dax.ko	dev_dax_split	struct vm_area *vma	PointerArg
		dev_dax_fault, dev_dax_pagesize	struct vm_fault *vmf	PointerArg
	dax.ko	dax_destroy_inode, dax_free_inode	struct inode *inode	PointerArg
DAX_OP*	nd_pmem.ko	pmem_dax_direct_access	struct dax_device *dax_dev	PointerArg
			pgoff_t pgoff	TaintedArgs
			long nr_pages	TaintedArgs
BLK_OP*	nd_pmem.ko	pmem_rw_page	struct block_device *bdev	PointerArg
			sector_t sector	TaintedArgs
	nd_btt.ko	btt_rw_page	struct page *page	TaintedArgsData
			unsigned int op	TaintedArgs
GETGEO*	nd_btt.ko	btt_getgeo	struct block_device *bdev	TaintedArgs
MOUNT*	dax.ko	dax_mount	struct file_system_type *fs_type	TaintedArgsData
IOCTL	libnvdimm.ko	nd_ioctl, nvdimm_ioctl	struct file *file	PointerArgs
			unsigned long arg	TaintedArgs

The first five types (*) are newly added entry types. The table also shows the specific functions, arguments, and taint types applied for individual arguments.

Table 9. Summary of Warnings Reported by Dr. Checker+ on Analyzing Five PM Kernel Modules

Detectors of Dr.Checker+	PM Kernel Modules					Total
	nfit.ko	nd_pmem.ko	nd_btt.ko	libnvdimm.ko	dax.ko	
IntegerOverflowDetector	0	4	2	0	0	6
TaintedPointerDereferenceChecker	0	0	4	0	0	4
GlobalVariableRaceDetector	1	0	0	12	2	15
TaintedSizeDetector	0	0	0	4	0	4
Total	1	4	6	16	2	29

result in information leak or buffer overflows. Table 9 summarizes the experimental results. Overall, Dr. Checker+ can process all the target kernel modules successfully, and we have observed warnings (i.e., potential issues) reported by its four detectors in five of the seven kernel modules (i.e., nfit.ko, nd_pmem.ko, nd_btt.ko, libnvdimm.ko, and dax.ko). For example, the *TaintedPointerDereferenceChecker* was able to identify a potential null pointer dereference in the nd_btt driver module, where the pointer variable bdev in the btt_getgeo entry point of nd_btt was accessed without a check for its validity. If the routines invoking the entry point function do not check for the null value before using it, then this code may lead to a kernel crash.

Note that the warnings reported by Dr.Checker+ do not necessarily imply PM bugs due to the conservative static analysis used in Dr.Checker. For example, the *GlobalVariableRaceDetector* in Dr. Checker would falsely report a warning for any access to a global variable outside of a critical section. By looking into the warnings reported in our experiments, we observed a similar false alarm: Dr. Checker+ may report a warning when the driver code invokes the macro "WARN_ON" to print to the kernel error log, which is benign.

Also, since Dr.Checker's detectors are stateless, they may report a warning for every occurrence of the same issue. For example, the page offset in pmem_dax_direct_access is used to calculate the physical address of a page, which involves bit manipulation operations and the resulting value may overflow the range of possible integer values. The *IntegerOverflowDetector* may report a warning whenever the method is invoked.

Overall, our experience is that extending Dr.Checker for analyzing PM-related issues in the Linux kernel is non-trivial and time-consuming due to the complexity of the PM subsystem.

However, the actual code modification is minor: We only need to modify about 100LoC in Dr.Checker to cover the major PM driver modules. While the effectiveness is still fundamentally limited by the capability of the core techniques used in the existing tools (e.g., Dr.Checker's static analysis may report false alarms), we believe that extending existing tools to make them work for the PM subsystem in the kernel can be an important first step toward addressing the PM-related challenges exposed in our study. We leave the investigation of improving Dr.Checker further (e.g., improving the static analysis algorithms, adding diagnosis support for understanding the root causes of warnings, or fixing the warnings detected) and building new detection tools as future work.

8 RELATED WORK

Studies of Software Bugs. Many researchers have performed empirical studies on bugs in open source software [39, 43, 49, 54, 68, 72, 78, 80, 83, 84, 107, 109]. For example, Lu et al. [80] studied 105 concurrency bugs from four applications and found that atomicity-violation and order-violation are two common bug patterns; Lu et al. [78] studied 5,079 file system patches (including 1,800 bugs fixed between Dec. 2003 and May 2011) and identified the trends of six file systems; Mahmud et al. [83, 84] studied bug patterns in file systems and utilities and extracted configuration dependencies that may affect the manifestation of bugs. Our study is complementary to the existing ones as we focus on bugs related to the latest PM technology, which may involve issues beyond existing foci (e.g., user-level concurrency bugs [49, 80, 109, 110], non-PM file systems [78], cryptographic modules [68], and configurations [83, 84]).

Studies of Production System Failures. Researchers have also studied various failure incidents in production systems [32, 55–57, 59, 60, 75, 96, 97, 108], many of which are caused by software bugs. For example, Gunawi et al. [55] studied 597 cloud service outages and derived multiple lessons including the outage impacts, causes, and so on; they found that many root causes were not described clearly. Similarly, Liu et al. [75] studied hundreds of high-severity incidents in Microsoft Azure. Due to the nature of the data source, these studies typically focus on high-level insights (e.g., caused by hardware, software, or human factors) instead of source-code level bug patterns described in this study. Since PM-based servers are emerging for production systems [30], and many production systems are based on Linux kernel, our study may help understand PM-related incidents in the real world.

Tools for Testing and Debugging Storage Systems. Many tools have been created to test storage systems [31, 36, 37, 51, 59, 83, 86, 87, 102, 111, 112] or help debug system failures [58, 64, 65, 71, 90, 104, 105, 108]. For example, EXPLODE [102], B^3 [87], and Zheng et al. [111] apply fault injections to emulate crash images to detect crash-consistency bugs in file systems, which has also been observed in our dataset. PFault [36, 59] applies fault injection to test Lustre and BeeGFS parallel file systems building on top of the Linux kernel. Gist [65] applies program slicing to help pinpoint the root cause of failures in storage software including SQLite, Memcached, and so on. Duo et al. [108] extended PANDA [48] to track device commands to help diagnosis. In general, these tools rely on detailed understanding of the underlying bug patterns to be effective. We believe our study on PM-related issues can contribute to building more effective bug detectors or debugging tools for PM-based storage systems, which we leave as future work.

9 DISCUSSIONS

9.1 Importance of Empirical Studies on Real-world Bug Patterns

As briefly mentioned in previous sections, this work was inspired by many existing research efforts on studying the characteristics of bugs in real-world software systems, including bug

patterns in multi-threaded applications [80], Linux and/or OpenBSD kernels [39, 43], (non-PM) file systems [78] and utilities [84], cryptographic software [68], cloud systems [54], and so on. Due to the complexity of real-world systems, it is practically impossible to build effective or efficient tools to address the issues induced by various bugs without thorough understanding of the bug characteristics. Therefore, such empirical studies of real-world bug patterns, which typically requires substantial manual efforts and domain knowledge, often serve as the foundation for understanding the issues and deriving practical solutions later. For example, the seminal work of S. Lu et al. [80] studied the concurrency bug patterns from four applications; while the study did not build any tools to address the bugs directly, it has inspired a series of follow-up research on concurrency bug detection and improving multi-threaded software in general [47, 50, 81, 94]. Similarly, the study of Linux file system patches by L. Lu et al. [78] has exposed general bug patterns in the context of file systems and has inspired various follow-up innovations on improving file systems reliability [51, 86]. One common lesson learned from these classic works is that "history repeats itself" and "learning from mistakes" is important for a better future [78, 80].

Our study follows a similar methodology (Section 3) but focuses on a dataset that has never been covered by existing works (to the best of our knowledge). As discussed in Section 5, we have identified a wide range of PM-related bug patterns, some of which are consistent to previous findings but others are not. For example, our study shows that cache-related issues only contribute to a small subset (1.7%) of the bugs in our dataset, and crash-consistency issues may be caused by semantic and concurrency bugs besides misusing low-level instructions. This observation is in contrast to recent research focus on PM bug detection in the community that mostly only consider crash-consistency issues caused by the misuse of cacheline flush instructions [53, 76, 77, 88]. In other words, additional research efforts are likely needed to address the PM-related issues in general.

As exemplified by previous studies, we envision multiple directions for follow-up research based on our empirical study of PM bug patches. First, the detailed characterisation of PM bug patterns (Section 5) may help derive concrete rules for building new PM bug detection tools, similar to our extension to Dr. Checker (Section 7.3). Second, the critical conditions identified in our reproducibility experiments (Section 6) may guide the design of future tools that need to consider how to trigger the bugs for bug detection or failure diagnosis. Third, the curated dataset of various types of PM bugs may serve as the test cases for evaluating the effectiveness of newly built tools, similarly to the concept of BugBench [79, 108]. We hope that by publishing our study results and releasing the dataset, the work can facilitate building the next generation of more robust PM-based storage systems.

9.2 Importance beyond Intel Optane DCPMM

As mentioned in Section 1 and Section 2, PM technologies are not limited to Intel Optane DCPMM [10], and the PM ecosystem in Linux is not solely designed for Intel. Various other PM technologies (e.g., PCM [95], STT-RAM [62], and CXL-based PM [34, 91]) will likely require similar support from the Linux kernel (e.g., the DAX feature). Therefore, our study of PM-related issues in the Linux kernel is not limited to Intel Optane technology either.

While Intel is winding down its Optane DCPMM business and will only provide limited support in the next 3 to 5 years [29], we expect to see other PM devices in the near future that will be largely compatible with the Linux PM ecosystem. Particularly, CXL [45] is a new open standard cache-coherent interconnect for processors, memory expansion, and accelerators. CXL 2.0 specification describes that PM devices can be realized using CXL.io and CXL.mem protocols [91]. Hence, it is expected that future PM devices would reside on CXL slots instead of memory DIMM slots. Although this change may impact how the host system recognizes PM devices, it is expected to

have little impact on the PM software stack. For example, CXL-PM driver code in the Linux kernel (`drivers/cxl/pmem.c`) relies on existing NVDIMM driver code for functions such as *register*, *un-register*, and *probe* devices [74]. CXL-based PM devices would still appear as special memory devices (similarly to existing PM devices), and existing software interfaces (e.g., DAX) are expected to work without any modifications. Therefore, our study on the current Linux PM subsystem, which is the foundation for supporting the next generation of CXL-based PM devices, is still relevant. However, we expect that CXL-based devices will likely introduce new CXL-specific drivers and specifications, which will add to the complexity of Linux PM subsystem and may introduce new bug patterns. Therefore, additional studies will likely be needed in the future.

10 CONCLUSIONS AND FUTURE WORK

This article presented a comprehensive study on PM-related patches and bugs in the Linux kernel. Based on 1,553 PM-related kernel patches, we derived multiple insights in terms of patch categories, bug patterns, consequences, fix strategies, and so on. We also performed reproducibility experiments to identify the critical configuration conditions for manifesting the bug cases. Moreover, we conducted tool extension experiments to explore the remedy solutions to address the issues exposed in our study. In the future, we plan to investigate new methodologies, including extensions to other tools, to address PM-related issues in the Linux PM ecosystem based on the detailed bug patterns identified in this work. More importantly, we hope that by sharing our study and releasing the characterized dataset, our efforts could facilitate follow-up research in the community and contribute to the development of effective PM bug detection tools and the enhancement of PM-based systems in general.

ACKNOWLEDGMENTS

We thank the TOS reviewers and editors for their insightful and constructive feedback. We also thank researchers from Western Digital, including Adam Manzanares, Filip Blagojevic, Qing Li, and Cyril Guyot, for valuable discussions on PM technologies. In addition, we thank Prakhar Bansal and Joshua Kalyanapu for running experiments on PM bug detectors. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES

[1] Advanced Configuration and Power Interface Specification. Retrieved from https://uefi.org/sites/default/files/resources/ACPI_6_1.pdf
[2] AGIGA KOMODO NVDIMM-N. Retrieved from http://agigatech.com/wp-content/uploads/2021/06/Unigen_Enterprise_Product-Brief_NVDIMM_DRAM_KOMODO_v2.2.pdf
[3] Classification of PM Bug Cases. Retrieved from https://git.ece.iastate.edu/data-storage-lab/prototypes/pm-bugs
[4] Computeexpresslink (CXL). Retrieved from https://www.computeexpresslink.org/
[5] dax: Convert to bitmask for Flags. Retrieved from https://patchwork.kernel.org/project/linux-fsdevel/patch/149875885239.10031.8327478660509602792.stgit@dwillia2-desk3.amr.corp.intel.com/
[6] DAX: Page Cache Bypass for Filesystems on Memory Storage. Retrieved from https://lwn.net/Articles/618064/
[7] HPE NVDIMM-N. Retrieved from https://support.hpe.com/hpesc/public/docDisplay?docId=c05302373
[8] HPE NVDIMM-N Drivers for Microsoft Windows. Retrieved from https://support.hpe.com/hpsc/swd/public/detail?swItemId=MTX_a77a79d838194d6498f355f2e4
[9] IBM eXFlash DIMM. Retrieved from https://www.ibm.com/support/pages/exflash-dimm-configuration-and-support-requirements-system-x
[10] Intel Optane DC Persistent Memory. Retrieved from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
[11] JEDEC. Retrieved from https://www.jedec.org
[12] Kernel.org Bugzilla. Retrieved from https://bugzilla.kernel.org/
[13] Kmemcheck. Retrieved from https://www.kernel.org/doc/html/v4.14/dev-tools/kmemcheck.html

[14] LIBNVDIMM: Non-Volatile Devices. Retrieved from https://www.kernel.org/doc/Documentation/nvdimm/nvdimm.txt
[15] LKL: Linux Kernel Library. Retrieved from https://lkl.github.io/
[16] Managing Persistent Memory. Retrieved from https://lrita.github.io/images/posts/filesystem/Managing-Persistent-Memory_0.pdf
[17] ndctl. Retrieved from https://github.com/pmem/ndctl
[18] NVM Programming Guide. Retrieved from https://www.snia.org/tech_activities/standards/curr_standards/npm
[19] Persistent Memory Development Kit (PMDK). Retrieved from https://pmem.io/pmdk/
[20] Persistent Memory FAQ. Retrieved from https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-faq.html
[21] PIN—A dynamic binary instrumentation tool. Retrieved from https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool/
[22] Storage Networking Industry Association. Retrieved from https://www.snia.org/
[23] Submitting Patches: The Essential Guide to Getting Your Code into the Kernel. Retrieved from https://www.kernel.org/doc/html/latest/process/submitting-patches.html
[24] syzkaller-kernel fuzzer. Retrieved from https://github.com/google/syzkaller
[25] The Kernel Address Sanitizer (KASAN). Retrieved from https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html
[26] The LLVM Compiler Infrastructure. Retrieved from https://llvm.org/
[27] The Undefined Behavior Sanitizer (UBSAN). Retrieved from https://www.kernel.org/doc/html/v4.9/dev-tools/ubsan.html
[28] ULLtraDIMM SSD Overview. Retrieved from https://www.snia.org/sites/default/files/SanDisk20ULLtraDIMM_0.pdf
[29] Update on PMDK and Our Long Term Support Strategy. Retrieved from https://pmem.io/blog/2022/11/update-on-pmdk-and-our-long-term-support-strategy/
[30] Dell EMC DCPMM: User's Guide. 2021. https://dl.dell.com/topicspdf/dcpmm-user-guide_en-us.pdf
[31] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C. Eric Schrock. 2013. Janus: Optimal flash provisioning for cloud storage workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*.
[32] Yehia Arafa, Atanu Barai, Mai Zheng, and Abdel-Hameed A. Badawy. 2018. Fault tolerance performance evaluation of large-scale distributed storage systems HDFS and ceph case study. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'18)*.
[33] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*.
[34] Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. 2022. Cache-coherent accelerators for persistent memory crash consistency. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*.
[35] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*.
[36] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. 2018. PFault: A general framework for analyzing the reliability of high-performance parallel file systems. In *Proceedings of the International Conference on Supercomputing (ICS'18)*. 1–11. https://doi.org/10.1145/3205289.3205302
[37] Jinrui Cao, Simeng Wang, Dong Dai, Mai Zheng, and Yong Chen. 2016. A generic framework for testing parallel file systems. In *Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS'16)*. IEEE Press, Los Alamitos, CA, 49–54. https://doi.org/10.1109/PDSW-DISCS.2016.12
[38] Eduardo Berrocal Garcia De Carellan. 2018. Discover Persistent Memory Programming Errors with Pmemcheck. Retrieved from https://software.intel.com/content/www/us/en/develop/articles/discover-persistent-memory-programming-errors-with-pmemcheck.html
[39] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys'11)*.
[40] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*.
[41] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. 2018. HiNFS: A persistent memory file system with both buffering and direct-access. *ACM Trans. Stor.* 14, 1 (2018), 1–30.

[42] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.* 30, 1 (2012), 1–49.
[43] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*.
[44] Leon Chua. 1971. Memristor—The missing circuit element. *IEEE Trans. Circ. Theory* 18, 5 (1971), 507–519.
[45] CXL Consortium. CXL Consortium. Retrieved from https://www.computeexpresslink.org/
[46] Intel Corporation. 2020. Intel Optane Persistent Memory Module: DSM Specification v2.0. Retrieved from https://pmem.io/documents/IntelOptanePMem_DSM_Interface-V2.0.pdf
[47] Dongdong Deng, Wei Zhang, and Shan Lu. 2013. Efficient concurrency-bug detection across inputs. *ACM Sigplan Not.* 48, 10 (2013), 785–802.
[48] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. 2015. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW'15)*.
[49] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2011. 2ndStrike: Toward manifesting hidden concurrency typestate bugs. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI'11)*.
[50] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2011. 2ndStrike: Toward manifesting hidden concurrency typestate bugs. *ACM Sigplan Not.* 46, 3 (2011), 239–250.
[51] Om Rameshwar Gatla, Muhammad Hameed, Mai Zheng, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojević, Cyril Guyot, and Robert Mateescu. 2018. Towards robust file system checkers. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*.
[52] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2019. Software wear management for persistent memories. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*.
[53] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. 415–428.
[54] Haryadi S. Gunawi, Thanh Do, Agung Laksono, Mingzhe Hao, Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, and Riza O. Suminto. 2014. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*.
[55] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why does the cloud stop computing? Lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*.
[56] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. 2018. Fail-slow at Scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Stor.* 14, 3 (2018), 1–26.
[57] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. 2013. Failure recovery: When the cure is worse than the disease. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS'13)*.
[58] Runzhou Han, Suren Byna, Houjun Tang, Bin Dong, and Mai Zheng. 2022. PROV-IO: An I/O-centric provenance framework for scientific data on HPC systems. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'22)*.
[59] Runzhou Han, Om Rameshwar Gatla, Mai Zheng, Jinrui Cao, Di Zhang, Dong Dai, Yong Chen, and Jonathan Cook. 2022. A study of failure recovery and logging of high-performance parallel file systems. *ACM Trans. Stor.* 18, 2 (2022), 1–44. https://doi.org/10.1145/3483447
[60] Runzhou Han, Duo Zhang, and Mai Zheng. 2020. Fingerprinting the checker policies of parallel file systems. In *Proceedings of the IEEE/ACM 5th International Parallel Data Systems Workshop (PDSW'20)*. https://doi.org/10.1109/PDSW51947.2020.00013
[61] Michael Hennecke. 2020. DAOS: A scale-out high performance storage stack for storage class memory. *Supercomput. Front.* (2020), 40.
[62] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, et al. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. In *Proceedings of the IEEE International Electron Devices Meeting (IEDM'05)*. 459–462.
[63] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*.
[64] Saisha Kamat, Abdullah Raqibul Islam, Mai Zheng, and Dong Dai. 2023. FaultyRank: A graph-based parallel file system checker. In *Proceedings of the 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS'23)*.

[65] Baris Kasikci, Benjamin Schubert, Cristiano L. Pereira, Gilles A. Pokam, and George Candea. 2015. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*.

[66] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*.

[67] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*.

[68] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why does cryptographic software fail? A case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys'14)*.

[69] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*.

[70] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB'19)*.

[71] Li Li, Yunhao Bai, Xiaorui Wang, Mai Zheng, and Feng Qin. 2017. Selective checkpointing for minimizing recovery energy and efforts of smartphone apps. In *Proceedings of the 8th International Green and Sustainable Computing Conference (IGSC'17)*.

[72] Li Li, Bruce Beitman, Mai Zheng, Xiaorui Wang, and Feng Qin. 2017. eDelta: Pinpointing energy deviations in smartphone apps via comparative trace analysis. In *Proceedings of the 8th International Green and Sustainable Computing Conference (IGSC'17)*.

[73] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System support for Improving Software Dependability (ASID'06)*. 25–33.

[74] Inc. Linux Kernel Organization. CXL-PMEM Driver. Retrieved from https://github.com/torvalds/linux/blob/master/drivers/cxl/pmem.c

[75] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What bugs cause production cloud incidents? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS'19)*.

[76] 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 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[77] 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 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

[78] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A study of linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*.

[79] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, Vol. 5.

[80] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*.

[81] Shan Lu, Soyeon Park, and Yuanyuan Zhou. 2011. Detecting concurrency bugs from the perspectives of synchronization intentions. *IEEE Trans. Parallel Distrib. Syst.* 23, 6 (2011), 1060–1072.

[82] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. 1007–1024.

[83] Tabassum Mahmud, Om Rameshwar Gatla, Duo Zhang, Carson Love, Ryan Bumann, and Mai Zheng. 2023. ConfD: Analyzing configuration dependencies of file systems for fun and profit. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*.

[84] Tabassum Mahmud, Duo Zhang, Om Rameshwar Gatla, and Mai Zheng. 2022. Understanding configuration dependencies of file systems. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. 1–8.

[85] Ashlie Martinez and Vijay Chidambaram. 2017. CrashMonkey: A framework to systematically test file-system crash consistency. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'17)*.

[86] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*.

[87] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*.

[88] 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 *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[89] Kevin P. O'Leary. 2018. How to Detect Persistent Memory Programming Errors Using Intel Inspector-Persistence Inspector. Retrieved from https://software.intel.com/content/www/us/en/develop/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.html

[90] Andrew Quinn, Jason Nelson Flinn, and Michael Cafarella. 2019. You can't debug what you can't see: Expanding observability with the OmniTable. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*.

[91] Andy Rudoff, Chet Douglas, and Tiffany Kasanicky. 2021. Persistent Memory in CXL. Retrieved from https://www.snia.org/educational-library/persistent-memory-cxl-2021

[92] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing configuration changes in context to prevent production failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 735–751.

[93] Paul Von Behren. 2015. NVML: Implementing persistent memory applications. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*.

[94] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. 2022. Controlled concurrency testing via periodical scheduling. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*.

[95] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[96] Erci Xu, Mai Zheng, Feng Qin, Jiesheng Wu, and Yikang Xu. 2018. Understanding SSD reliability in large-scale cloud systems. In *Proceedings of the 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW'18)*.

[97] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. 2019. Lessons and actions: What we learned from 10K SSD-related storage system failures. In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*.

[98] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*.

[99] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. KRace: Data race fuzzing for kernel file systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*.

[100] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.

[101] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*.

[102] Junfeng Yang, Can Sar, and Dawson Engler. 2006. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 131–146.

[103] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*.

[104] Di Zhang, Dong Dai, Runzhou Han, and Mai Zheng. 2021. SentiLog: Anomaly detecting on parallel file systems via log-based sentiment analysis. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'21)*. 86–93.

[105] Di Zhang, Chris Egersdoerfer, Tabassum Mahmud, Mai Zheng, and Dong Dai. 2023. Drill: Log-based anomaly detection for large-scale storage systems using source code analysis. In *Proceedings of the 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS'23)*.

[106] Duo Zhang, Om Rameshwar Gatla, Wei Xu, and Mai Zheng. 2021. A study of persistent memory bugs in the linux kernel. In *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR'21)*.

[107] Duo Zhang, Tabassum Mahmud, Om Rameshwar Gatla, RunZhou Han, Yong Chen, and Mai Zheng. 2022. On the reproducibility of bugs in file-system aware storage applications. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS'22)*.

[108] Duo Zhang and Mai Zheng. 2021. Benchmarking for observability: The case of diagnosing storage failures. In *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 1, 1 (2021).

[109] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: A low-overhead mechanism for detecting data races in GPU programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*.

[110] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2014. GMRace: Detecting data races in GPU programs via a low-overhead scheme. *IEEE Trans. Parallel Distrib. Syst.* 25, 1 (2014).
[111] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.
[112] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*.
[113] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. 2017. Reliability analysis of SSDs under power fault. *ACM Trans. Comput. Syst.* (2017). http://dx.doi.org/10.1145/2992782
[114] Jiang Zhou, Yong Chen, Mai Zheng, and Weiping Wang. 2023. Data distribution for heterogeneous storage systems. *IEEE Trans. Comput.* 72, 6 (2023).

Received 9 July 2022; revised 12 March 2023; accepted 9 May 2023