A Study of Persistent Memory Bugs in the Linux Kernel

Duo Zhang*, Om Rameshwar Gatla*, Wei Xu, Mai Zheng
Department of Electrical and Computer Engineering, Iowa State University
<duozhang,ogatla,weixu,mai>@iastate.edu

ABSTRACT

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 paper presents a comprehensive study of PM-related bugs in the Linux kernel. By analyzing 1,350 PM-related kernel patches in depth, we derive multiple insights in terms of PM patch categories, PM bug patterns, consequences, and fix strategies. We hope our results could contribute to the development of effective PM bug detectors and robust PM-based systems.

CCS CONCEPTS

• Software and its engineering → Operating systems; **• Hardware** → *Memory and dense storage.*

KEYWORDS

Persistent Memory, Kernel Patches, Bug Detection, Reliability

ACM Reference Format:
Duo Zhang*, Om Rameshwar Gatla*, Wei Xu, Mai Zheng. 2021. A Study of Persistent Memory Bugs in the Linux Kernel . In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3456727.3463783

1 INTRODUCTION

Persistent memory (PM) technologies offer attractive features for developing storage systems and applications. For example, Intel® Optane™ PM [6] can support byte-granularity accesses with latencies less than 3x of DRAM latencies [49], while also providing durability guarantees. Such new properties have inspired a wide range of PM-based software optimizations [5, 20, 24, 34, 46].

Unfortunately, building correct PM-based software systems is challenging [33, 43]. 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 [11]), multiple OS kernel subsystems must be revised (e.g., `dax`, `libnvdimm`). Such complexity could potentially lead to obscure bugs that hurt system reliability.

* Both authors contributed equally

SYSTOR '21, June 14–16, 2021, Haifa, Israel
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8398-1/21/06.
https://doi.org/10.1145/3456727.3463783

Addressing the challenge above will require cohesive efforts from multiple related directions including PM bug detection [18, 36, 37, 41, 42], PM programming support [11], PM specifications [23], 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 [19, 22, 25, 32, 38, 39]. For example, Lu *et al.* [38] studied 5,079 patches of 6 Linux file systems and derived various patterns of file system evolution. Neal *et al.* [41] studied 63 PM bugs (mostly from PMDK [11]) 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 due to the different foci.

In this paper, 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 [8, 10, 12]. Our study is based on 1,350 PM-related patches committed in Linux between Jan. 2011 and Dec. 2020. 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 (39.9%) are for adding new features or improving the efficiency of existing ones, and a similar portion (37.1%) are for maintenance. 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 (23.0%) 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`', '`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 (47.2% and 16.2% respectively), although the root causes are different. Also, many PM bugs are uniquely dependent on hardware (20.5%), which may be caused by misunderstanding of specifications, miscalculation of addresses, etc. 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 [38]. Also, 21.6% 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. On the other hand, we observe that different PM bugs may be fixed in a similar way by refining the sanity checks.

In addition, we look into 3 representative PM bug detectors [36, 37, 41] and find that they are largely inadequate for addressing the

Category	Description
Bug (23.0%)	Fix existing correctness issues (e.g., misalignment of PM regions, race on PM pages)
Feature (39.9%)	Add new features or improve efficiency of existing ones (e.g., extend device flags, reduce write overhead)
Maintenance (37.1%)	Polish source code, compilation scripts, and documentation (e.g., delete obsolete code, fix compilation errors)

Table 1: Three Categories of PM-related Patches.

PM bugs in our study. On the other hand, a few recently proposed non-PM bug detectors [29, 30, 47, 48] could potentially be applied to detect a great portion of PM bugs if one common challenge is addressed. We hope our study and the resulting dataset [2] could contribute to the development of effective PM bug detectors and the enhancement of robust PM-based systems.

The rest of the paper is organized as follows: §2 describes the study methodology; §3 presents the overview of PM patches; §4 characterizes PM bugs in details; §5 discusses the implications on bug detection; §6 discusses related work and §7 concludes the paper.

2 METHODOLOGY

In this section, we describe how we collect the dataset for study (§2.1), how we characterize the PM-related patches and bugs (§2.2), and the limitations of the methodology (§2.3).

2.1 Dataset Collection and Refinement

All changes to the Linux kernel occur in the form of patches [14], including but not limited to bug fixes. We collect PM-related patches from the Linux source tree for study via three steps as follows:

First, we collect all patches committed to the Linux kernel between Jan. 2011 and Dec. 2020, which generates a dataset containing more than 693,000 patches.

Second, in order 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`', etc. The resulting dataset contains 2,541 patches. Note that this step is similar to the keyword search in previous studies [26, 39].

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 1,350 PM-related patches.

2.2 Dataset Analysis

Based on the 1,350 PM-related patches, we conduct a comprehensive study to answer three set of questions:

- 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?
- 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 [14], 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. We present our findings for the three sets of questions above in §3, §4, and §5, respectively.

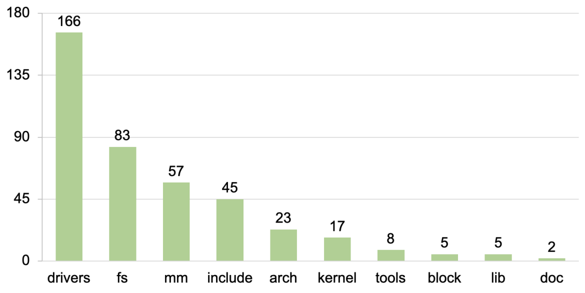

Figure 1: Counts of PM Bug Patches in the Kernel Source Tree.

2.3 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 [2].

3 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 updates.

The largest category is '`Feature`' (39.9%), which is different from previous studies where maintenance patches tend to be dominant [38]. This reflects the significant changes needed to add PM to the Linux ecosystem which 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"* [3]), which may imply that PM-based extensions tend to be well-planned in advance.

The second largest category is '`Maintenance`' (37.1%), which reflects the significant effort needed to keep PM-related kernel components well-maintained.

The '`Bug`' patches, which directly represent correctness issues in the kernel, account for a non-negligible portion (23.0%), We analyze this important set of patches further in the next section.

4 PM BUG CHARACTERISTICS

4.1 Where Are the Bugs

Figure 1 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`'

File Name	# of Occur.	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

Table 2: Top 10 Most "Buggy" Files

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 minor 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

Table 3: Classification of PM Bug Patterns. ***The last column shows the major subsystems (up to 3) affected by the bugs.***

for file systems, 'mm' for memory management). In case a patch modifies multiple files across different directories (which is not uncommon as will be discussed in §4.4.1), we count it towards 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 [22]. In the PM context, this is largely due to the introduction of `nvdimm` driver. Also, '`fs`' accounts for the second most patches, largely due to the complexity of adding `dax` support for file systems [4]. 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` supports are the two major sources of introducing PM bugs in the kernel.

4.2 Bug Pattern

To build robust PM systems, it is important to understand the types of bugs occurred. We analyze the PM bug patches in depth and classify the bugs into five types (Table 3): *Hardware Dependent*, *Semantic*, *Concurrency*, *Memory* and *Error Code*. Each type includes multiple subtypes. The last column of Table 3 shows the major subsystems affected by each type of bugs. For clarity, the column only lists up to 3 subsystems for each type. We can see that the same type of bugs may affect multiple subsystems (e.g. '`drivers`', '`fs`', '`mm`'). The percentages of the five types and the subtypes are shown in Figure 2 and Figure 3, respectively. Due to space limits, we only discuss a few representative cases below.

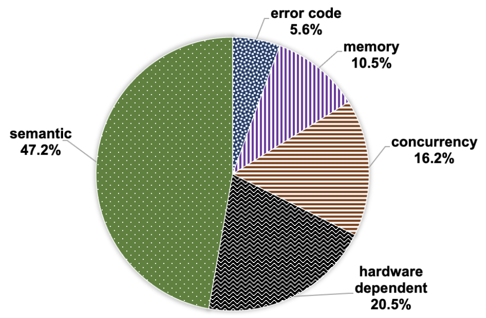

Figure 2: Percentages of PM Bug Types

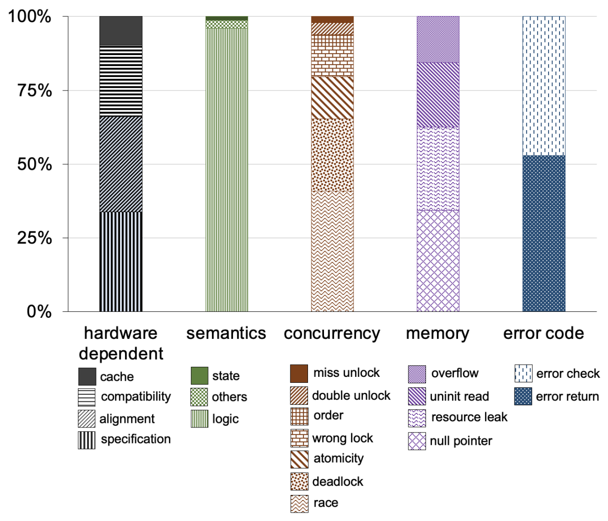

Figure 3: Percentages of PM Bug Subtypes

Compared to previous studies [38, 39], the most unique pattern observed in our dataset is *Hardware Dependent*, which accounts for 20.5% of PM bugs (Figure 2). There are four subtypes including *Specification* (33.9% of *Hardware Dependent*), *Alignment* (32.3%), *Compatibility* (24.2%), and *Cache* (9.6%), which reflects four different aspects of challenge for integrating PM devices correctly to the Linux kernel.

Specification is the largest subtype of *Hardware Dependent* bugs (33.9%). Figure 4 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.

drivers/nvdimm/bus.c

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

Figure 4: A Specification Bug Example. ***This bug was caused by the ambiguity of the ACPI specification.***

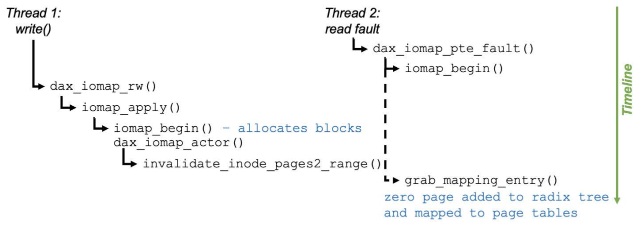

Figure 5: A Concurrency Bug Example. ***This bug was caused by a race condition involving DAX IO operations.***

In terms of the other 3 subtypes, we find that *Alignment* issues are typically caused by the inconsistency between various abstractions of PM devices (e.g., PM regions, namespaces); *Compatibility* issues often arise when the new dax functionality conflicts with the underlying CPU architecture or PM device; *Cache* bugs are caused by misuse of cache-related operations (e.g., `clflushopt`), which is the focus of previous studies [41]. However, the *Cache* subtype only accounts for 9.6% of *Hardware Dependent* bugs (Figure 3), which implies that kernel-level PM issues are more complex than the typical pattern observed in user-level programs.

In addition to *Hardware Dependent*, we find that PM bugs may follow the classic *Semantic*, *Concurrency*, *Memory*, *Error Code* patterns [38, 39], although the root causes are typically different due to the different contexts. For example, the *State* in *Semantic* are often related to PM device states instead of file system states [38]. Similarly, *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 fault handler and regular IO operations. Figure 5 shows one specific example involving two threads. In this case, Thread 1 invokes a write syscall which allocates blocks on PM, and Thread 2 invokes a read to the same PM blocks which 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. 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.

4.3 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 8 types of consequence, including *Missing Device*, *Inaccessible Device*, *Security*, *Corruption*, *Crash*, *Hang*, *Wrong Return Value*, and *Resource Leak*. We elaborate on the first three types as they are relatively more unique to PM (the others are similar to previous studies [38]):

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.

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.

In addition, we observe two *Security* issues. For example, write operations may be allowed on read-only `dax` mappings, which exposes wrong access permissions to the end user.

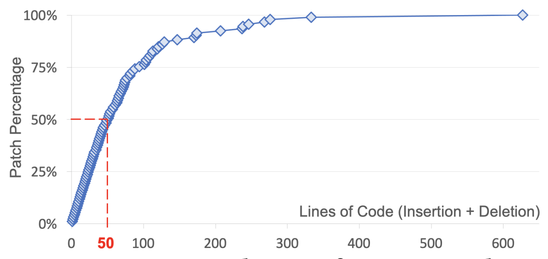

Figure 6: Size Distribution of PM Bug Patches

File Count	1	2	3	4	5	> 5
Patch %	64.0%	14.0%	8.8%	6.1%	2.9%	4.2%
Dir. Count	**1**	**2**	**3**	**4**	**5**	**> 5**
Patch %	78.4%	13.2%	5.8%	1.8%	0.6%	0.3%

Table 4: Scope of PM Bug Patches. ***This table shows the % of bug patch involving different counts of files or directories.***

4.4 Bug Fix

4.4.1 *How difficult it is to fix PM bugs.*

To quantitatively measure the complexity of fixing PM bugs, we calculate three metrics:

Bug Patch Size. We define the patch size as the sum of lines of insertion and deletion in the patch. Figure 6 shows the distribution of bug patch sizes. We can see that most bug patches are relatively small. For example, 50% patches have less than 50 lines of insertion and deletion code (LoC). However, compared to traditional non-PM file system bug patches where 50% are less than 10 LoC [38], 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 (64.0%) or files within one directory (78.4%). On the other hand, 4.2% patches may involve more than 5 files. Moreover, a non-negligible portion of patches involve more than one directories (21.6%). 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 the scope too. We observe that only 5% of the sampled GPU patches involve more than one directory, which is much less than the 21.6% cross-subsystem PM bug patches.

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 which may provide more complete time-to-fix information (e.g., Bugzilla [7]), which we leave as future work.

4.4.2 *Fix Strategy.*

We find that the strategies for fixing PM bugs often vary a lot depending on the specific bug types (Table 3). On the other hand, we also observe that different types of PM bugs may be fixed by one common strategy: refining sanity checks. For example, in one alignment bug case, 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`). 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.

5 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 21.6% PM bug patches involve multiple kernel subsystems, simply focusing on one subsystem is unlikely enough. On the other hand, since many bugs in different subsystems may follow similar patterns, capturing one bug pattern may benefit multiple subsystems. We further discuss the limitations of a few state-of-the-art bug detection tools as well as the opportunities in this section.

PM Bug Detectors: Multiple PM-specific bug detection tools have been proposed recently [36, 37, 41]. We are able to verify their effectiveness by reproducing most of the reported detection results. Unfortunately, we find that they are fundamentally limited for capturing the PM bugs in our dataset. For example, XFDetector [36] relies on Intel Pin [13] which can only instrument user-level programs. PMTest [37] can be applied to kernel modules, but it requires manual annotations which is impractical for major kernel subsystems. AGAMOTTO [41] relies on KLEE [17] 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 [21]), novel PM-specific path reduction algorithms are likely needed to avoid the state explosion problem [31].

Non-PM Bug Detectors: Great efforts have been made to detect non-PM bugs in the kernel [29, 30, 40, 47, 48]. For example, CrashMonkey [40] logs the `bio` requests and emulates crashed disk states to test the crash consistency of traditional file systems. Such crash consistency issues likely exist in PM subsystems too due to the complexity. 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 [29, 30, 47, 48]. For example, Razzer [29] combines fuzzing with static analysis and detects data races in multiple kernel subsystems (e.g., '`driver`', '`fs`', '`mm`'), which could potentially be extended to cover a large portion of concurrency PM bugs in our dataset. Since Razzer and similar fuzzers heavily rely on virtualized (e.g., QEMU [16]) or simplified (e.g., LKL [9]) environments to achieve high efficiency, one common challenge and opportunity for extending them is to emulate PM devices and interfaces precisely to ensure the fidelity, which we leave as future work.

6 RELATED WORK

Studies of Software Bugs. Many researchers have performed empirical studies on bugs in open source software [19, 22, 25, 32, 38, 39]. For example, Lu *et al.* [39] studied 105 concurrency bugs from 4 applications and found that atomicity-violation and order-violation are two common bug patterns; Lu *et al.* [38] studied 5,079 file system patches (including 1,800 bugs fixed between Dec. 2003 and May 2011) and identified the trends of 6 file systems. 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 [39], non-PM file systems [38]).

Studies of Production System Failures. Researchers have also studied various failure incidents in production systems [26–28, 35, 44, 45], many of which were caused by software bugs. For example, Gunawi *et al.* [26] studied 597 cloud service outages and derived multiple lessons including the outage impacts, causes, etc; Liu *et al.* [35] studied hundreds of incidents in Microsoft Azure. Due to the nature of the data source, these studies typically focus on high-level insights instead of source-code level bug patterns. Since PM-based servers are emerging for production systems [15], our study may help understand PM-related incidents in the real world.

7 CONCLUSIONS

This paper presented the first comprehensive study on PM-related patches and bugs in the Linux kernel. We hope our efforts could contribute to the development of effective PM bug detection tools and the enhancement of PM-based systems.

8 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and Ryan Stutsman (our shepherd) for their insightful feedback. We also thank researchers from Western Digital including Adam Manzanares, Filip Blagojevic, Qing Li, and Cyril Guyot for valuable discussions on PM technologies. This work was supported in part by NSF under grants CNS-1566554, CNS-1855565, and CNS-1943204. 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. https://uefi.org/sites/default/files/resources/ACPI_6_1.pdf.

[2] Classification of PM Bug Cases. https://git.ece.iastate.edu/data-storage-lab/prototypes/pm-bugs.

[3] dax: convert to bitmask for flags. https://patchwork.kernel.org/project/linux-fsdevel/patch/149875885239.10031.8327478660509602792.stgit@dwillia2-desk3.amr.corp.intel.com/.

[4] DAX: Page cache bypass for filesystems on memory storage. https://lwn.net/Articles/618064/.

[5] HPE NVDIMM-N Drivers for Microsoft Windows . https://support.hpe.com/hpsc/swd/public/detail?swItemId=MTX_a77a79d838194d6498f355f2e4.

[6] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[7] Kernel.org Bugzilla. https://bugzilla.kernel.org/.

[8] LIBNVDIMM: Non-Volatile Devices. https://www.kernel.org/doc/Documentation/nvdimm/nvdimm.txt.

[9] LKL: Linux Kernel Library. https://lkl.github.io/.

[10] Managing Persistent Memory. https://lrita.github.io/images/posts/filesystem/Managing-Persistent-Memory_0.pdf.

[11] Persistent Memory Development Kit (PMDK). https://pmem.io/pmdk/.

[12] Persistent Memory FAQ. https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-faq.html.

[13] PIN — a dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool/.

[14] Submitting Patches: The Essential Guide to Getting your Code into the Kernel . https://www.kernel.org/doc/html/latest/process/submitting-patches.html.

[15] Dell EMC DCPMM: User's Guide. https://dl.dell.com/topicspdf/dcpmm-user-guide_en-us.pdf, January, 2021.

[16] Fabrice Bellard. QEMU, A Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[17] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. 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)*, 2008.

[18] Eduardo Berrocal Garcia De Carellan. Discover Persistent Memory Programming Errors with Pmemcheck . https://software.intel.com/content/www/us/en/develop/articles/discover-persistent-memory-programming-errors-with-pmemcheck.html, 2018.

[19] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)*, 2011.

[20] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. HiNFS: A Persistent Memory File System with Both Buffering and Direct-Access. In *ACM Transactions on Storage (TOS)*, 2018.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, Implementation, and Applications. In *ACM Transactions on Computer Systems (TOCS)*, 2012.

[22] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[23] Intel Corporation. Intel Optane Persistent Memory Module: DSM Specification v2.0 . https://pmem.io/documents/IntelOptanePMem_DSM_Interface-V2.0.pdf, 2020.

[24] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Software Wear Management for Persistent Memories. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[25] Haryadi S. Gunawi, Thanh Do, Agung Laksono, Mingzhe Hao, Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, and Riza O. Suminto. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[26] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[27] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *ACM Transactions on Storage (TOS)*, 2018.

[28] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure Recovery: When the Cure is Worse than the Disease. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS)*, 2013.

[29] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[30] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[31] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[32] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does Cryptographic Software Fail? A Case Study and Open Problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys)*, 2014.

[33] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)* , 2019.

[34] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating Persistent Memory Range Indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, 2019.

[35] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What Bugs Cause Production Cloud Incidents? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.

[36] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 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)*, 2020.

[37] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 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)*, 2019.

[38] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[39] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 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)*, 2008.

[40] Ashlie Martinez and Vijay Chidambaram. 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)*, 2017.

[41] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How Persistent is your Persistent Memory Application? In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

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

[43] Paul Von Behren. NVML: Implementing Persistent Memory Applications. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[44] Erci Xu, Mai Zheng, Feng Qin, Jiesheng Wu, and Yikang Xu. 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)*, 2018.

[45] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and Actions: What we Learned from 10K SSD-related Storage System Failures. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[46] Jian Xu and Steven Swanson. 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)*, 2016.

[47] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRace: Data Race Fuzzing for Kernel File Systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[48] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, 2020.