Efficiently Detecting Concurrency Bugs in Persistent Memory Programs

Zhangyu Chen
Huazhong University of Science and Technology
Wuhan, Hubei, China
chenzy@hust.edu.cn

Yu Hua*
Huazhong University of Science and Technology
Wuhan, Hubei, China
csyhua@hust.edu.cn

Yongle Zhang
Purdue University
West Lafayette, Indiana, USA
yonglezh@purdue.edu

Luochangqi Ding
Huazhong University of Science and Technology
Wuhan, Hubei, China
lcqding@hust.edu.cn

ABSTRACT

Due to the salient DRAM-comparable performance, TB-scale capacity, and non-volatility, persistent memory (PM) provides new opportunities for large-scale in-memory computing with instant crash recovery. However, programming PM systems is error-prone due to the existence of crash-consistency bugs, which are challenging to diagnose especially with concurrent programming widely adopted in PM applications to exploit hardware parallelism. Existing bug detection tools for DRAM-based concurrency issues cannot detect PM crash-consistency bugs because they are oblivious to PM operations and PM consistency. On the other hand, existing PM-specific debugging tools only focus on sequential PM programs and cannot effectively detect crash-consistency issues hidden in concurrent executions.

In order to effectively detect crash-consistency bugs that only manifest in concurrent executions, we propose PMRace, the first PM-specific concurrency bug detection tool. We identify and define two new types of concurrent crash-consistency bugs: *PM Inter-thread Inconsistency* and *PM Synchronization Inconsistency*. In particular, PMRace adopts PM-aware and coverage-guided fuzz testing to explore PM program executions. For *PM Inter-thread Inconsistency*, which denotes the data inconsistency hidden in thread interleavings, PMRace performs PM-aware interleaving exploration and thread scheduling to drive the execution towards executions that reveal such inconsistencies. For *PM Synchronization Inconsistency* between persisted synchronization variables and program data, PMRace identifies the inconsistency during interleaving exploration. The post-failure validation reduces the false positives that come from custom crash recovery mechanisms. PMRace has found 14 bugs (10 new bugs) in real-world concurrent PM systems including PM-version memcached.

*Corresponding author

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507755

CCS CONCEPTS

• **Hardware → Memory and dense storage**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

Persistent Memory, Crash Consistency, Testing, Debugging, Concurrency

ACM Reference Format:
Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. 2022. Efficiently Detecting Concurrency Bugs in Persistent Memory Programs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507755

1 INTRODUCTION

Persistent Memory (PM) is a technology that provides large-scale non-volatile memory with DRAM-comparable performance. PM devices, such as Intel Optane DC PM [8] — a real PM product with up to 512 GB capacity per module, have been available on the market. Recent research [23, 62] show that PM significantly boosts the performance of many applications with instant crash recovery.

Programming PM systems is error-prone due to the existence of crash-consistency bugs [18, 34, 43, 44]. In typical write-back CPU caches, common PM writes are cached in the volatile CPU caches but not immediately flushed (persisted) to PM. Moreover, the order of cache flushes is not guaranteed to respect the PM write order because of potential reordering in cache eviction. However, PM data need to remain consistent or can be recovered after system restarts, called *crash consistency*. To ensure efficient crash consistency, developers need to insert appropriate cache line flushes (e.g., `CLWB`) and fences (e.g., `SFENCE`) after PM writes. This programming practice is error-prone and requires expertise in crash consistency. Though programmers can utilize the transaction interfaces from PM libraries, e.g., Persistent Memory Development Kit (PMDK) [10], such interfaces are typically implemented using write-ahead logging and introduce non-negligible overheads. The misuse of PM libraries also leads to crash-consistency bugs [43, 44].

Crash-consistency bugs in PM are extremely challenging to diagnose, especially for the concurrent PM systems exploiting hardware parallelism [3, 12, 31, 42, 69]. It prevents developers from performing architecture oblivious debugging and forces developers to

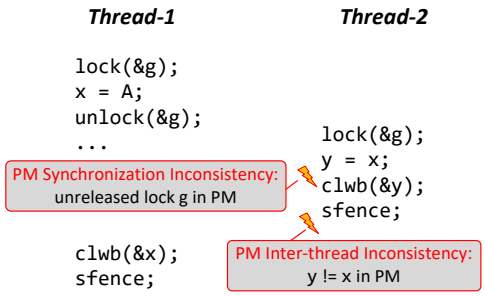


Figure 1: An example of PM concurrency bugs.

consider possible interleavings between cache flushes (persistency states) and program executions, similar to the way concurrency bugs [32, 37] force developers to consider possible interleavings among threads and processes. However, due to the lack of persistency awareness, existing DRAM-based concurrency bug detection tools cannot find PM-specific crash-consistency bugs involving thread interleavings.

We refer to concurrency bugs that only occur in PM programs (not triggered in non-PM programs) as *PM concurrency bugs*. Many PM-specific testing tools have been proposed to detect the violations of crash consistency in PM programs by leveraging various mechanisms, e.g., programmer-annotated assertions [36], crash-consistency bug patterns [14, 34, 35], symbolic execution [44], model checking [19, 28], and likely-correctness conditions with output equivalence checking [18], but none targets PM concurrency bugs. In fact, the concurrency bugs triggered only in specific interleavings are known to be more difficult to detect [32, 37].

In this paper, we target PM-specific concurrency bugs that have not been addressed in existing testing tools. In particular, we categorize PM concurrency bugs into two categories, identify one new type of inconsistency in each category that lead to PM concurrency bugs, explore the challenges to detect bugs caused by such inconsistencies, and build a fuzz testing tool to effectively identify PM concurrency bugs.

PM concurrency bugs break PM applications' crash consistency guarantees under concurrent executions. According to the types of broken consistency guarantees, we categorize PM concurrency bugs into two categories: *PM Interleaving Concurrency Bugs* happen when the interleaving of concurrent executions breaks the crash consistency guarantee on PM application's data such as program variables. *PM Execution Context Bugs* occur when the broken crash consistence guarantee involves a dynamic instance of concurrent execution context (e.g., thread state and program data). For instance, if a lock is acquired before a crash and restored to the locked state after restarts, the execution context of the corresponding thread holding the lock should also be recovered to a state consistent with the acquired lock (immediately before the crash). Otherwise, after restarts, the unreleased lock causes a hang when threads try to acquire this lock. For each category, we identify a new type of inconsistency that could lead to PM concurrency bugs.

PM Inter-thread Inconsistency: We define a program execution in which one thread makes durable side effects (e.g., writes to PM/disks) based on non-persisted data written by another thread as a *PM Inter-thread Inconsistency*. For example, as shown in Figure 1, thread-1 writes a value A to a shared variable x without immediately flushing the value to PM (via CLWB and SFENCE). Thread-2 reads non-persisted value A, writes the read value A to another variable y, and flushes y to PM. A crash-consistency bug occurs under the interleaving if a crash occurs after thread-2 flushes y and before thread-1 flushes x to PM. After recovery, y is not equal to x, thus causing crash inconsistency.

PM Synchronization Inconsistency: A synchronization variable refers to a piece of shared data, such as a lock and a mutex, that coordinates the execution of threads. Assuming threads (including their execution contexts such as program counter and stack) are not persisted to PM, if synchronization variables are persisted to PM and recovered after crashes, the inconsistency between synchronization variables and the new threads in the recovered PM application induces a new type of inconsistency, called *PM Synchronization Inconsistency*. For example, in Figure 1, if a crash occurs right after thread-2 gets and persists the lock g, after recovery all future accesses to variables x and y will be blocked due to the locked state of g.

Existing testing tools for sequential PM programs cannot detect *PM Inter-thread Inconsistency* or *PM Synchronization Inconsistency* because they do not check cross-thread crash inconsistency in thread interleavings and ignore the inconsistency between synchronization data and executing threads. Given the previous example, existing testing tools for persistency checking only automatically verify whether the writes to x and y are followed by cache flushes in each thread. Witcher [18] detects ordering violation via dependency analysis but only focuses on single-threaded PM programs.

We propose *PMRace*, a PM-aware coverage-guided fuzzer to efficiently detect consistency bugs for concurrent PM programs. Our PMRace addresses two main challenges.

False Positives. A false positive refers to the case that an identified bug is not a true bug. Finding all consistency bugs without false positives is challenging, since each PM system has application-specific data structures and unique consistency requirements. Moreover, the recovery mechanism in an application can recover from an inconsistent state, causing false positives in the detected inconsistencies. Output equivalence checking [18] avoids false positives but at the cost of false negatives (missing true bugs) — some implicit crash-consistency bugs that do not affect program's output, such as PM leakage and redundant flushes, successfully pass the output equivalence checking. In order to reduce the false positive rate of identified inconsistencies without missing true bugs, PMRace detects PM inconsistencies using fuzz testing with post-failure validation. For *PM Inter-thread Inconsistency*, PMRace instruments the target application and accurately captures the cross-thread data inconsistency by identifying reading non-persisted data and the following durable side effects via taint analysis. In the post-failure stage, PMRace automatically validates if the detected inconsistent data and user-annotated synchronization variables for *PM Synchronization Inconsistency* are correctly recovered to a consistent state. The detected inconsistencies not fixed by the immediate recovery code are marked as consistency bugs, and the detailed bug reports are attached for bug diagnosis.

Exponential Interleaving Search Space. For concurrent programs, each non-atomic instruction can be a preemption point and the possible interleavings grow at an exponential rate with respect

to the number of instructions in the application. Conventional interleaving exploration schemes are unaware of PM characteristics and cost-inefficient to detect PM-specific concurrency bugs. To accelerate the bug detection, PMRace leverages PM-aware coverage-guided interleaving exploration. We propose a new coverage metric, *PM alias pair coverage*, to record the tested thread interleavings. For interleaving exploration, PMRace performs PM-aware thread scheduling to drive the execution towards reading non-persisted data by injecting conditional waiting in prioritized preemption points, which are selected from the shared data accesses to PM.

Though our tool focuses on *PM Inter-thread Inconsistency* and *PM Synchronization Inconsistency*, these two inconsistencies are representative for PM concurrency bugs: *PM Inter-thread Inconsistency* represents *PM Interleaving Concurrency Bugs* and *PM Synchronization Inconsistency* represents *PM Execution Context Bugs*. Moreover, PMRace's framework is easy-to-use and extensible for other (concurrency) bug patterns by adding new PM checkers and interleaving exploration strategies. For instance, the checking for *PM Synchronization Inconsistency* can be leveraged to detect other inconsistencies between unreleased exclusive resources (e.g., sockets) and execution contexts. In the evaluation, we have implemented a random delay injection scheme using PMRace's framework for interleaving exploration in fuzz testing for comparison.

In summary, we have made the following contributions:

- **PM Concurrency Bug Patterns.** We identify and define new PM-specific concurrency bug patterns, including *PM Inter-thread Inconsistency* and *PM Synchronization Inconsistency*.
- **PM-Aware Coverage-Guided Fuzzing.** We design PM checkers and PM-aware coverage-guided fuzzing in PMRace. By using the PM-aware interleaving exploration and post-failure validation, PMRace efficiently identifies PM concurrency bugs. To the best of our knowledge, PMRace is the first testing tool for concurrent PM programs.
- **New Bugs.** We have implemented PMRace that is used to test real-world concurrent PM systems. PMRace has found 14 bugs, 10 of which are new bugs. Our source code is available at https://github.com/yhuacode/pmrace.

2 BACKGROUND AND MOTIVATION

2.1 Crash Consistency in PM Programming

Persistent memory (PM) provides byte-addressability and durability, thus enabling high performance and instant recovery for many applications. After memory-mapping, the data in PM can be directly accessed via byte-grained load/store instructions bypassing page cache, which delivers DRAM-scale performance [5, 27, 31, 62, 66]. Unlike DRAM, PM data survive power outages due to the inherent non-volatility of device characteristics (e.g., 3D XPoint [20]).

The non-volatility of PM introduces the crash consistency requirement for programming. Although PM is durable for stored data, typical CPU caches are volatile. Moreover, the persist order depends on the eviction order of cache lines (based on cache replacement strategies and manual flush operations), which is different from the issue order of store instructions. Due to the volatile CPU caches and write reordering, it is nontrivial to implement efficient and crash-consistent PM programs. Some programs leverage high-level mechanisms, e.g., logging [27, 66] and Copy-on-Write (CoW) [42, 69], to enforce crash consistency. To avoid the high costs in write-ahead logging or CoW, it is possible to use low-level instructions to explicitly flush cache lines for persistency and insert memory fences for ordering [12, 31, 42, 69]. For instance, the ISA in x86 systems [9] provides `CLWB` and `CLFLUSHOPT` to flush a cache line into (persistent) memory and `SFENCE` to ensure previous store or flush operations are visible before any store or flush operations following the `SFENCE`. Similar flush (e.g., `DC CVAP`) and fence (e.g., `DSB`) instructions are available in ARM [1, 33]. Alternatively, programmers can use non-temporal stores to directly write data to PM bypassing CPU caches [62]. However, it disables the fast caching in CPU caches and is unsuitable for hot data.

2.2 Fuzz Testing

Fuzzing is a common software testing technique interpreted as the process of running programs repeatedly with inputs from its generator. *Fuzz testing* refers to the use of fuzzing in testing. Each round of fuzz testing with a generated input is called a *fuzz campaign*. The tool for fuzz testing is called a *fuzzer*. Due to its simplicity and efficiency, fuzzing has been widely used in the security and system communities [34, 40, 41, 59]. Leveraging fuzz testing to check the correctness and efficiency of PM programs is promising to improve the software quality [34, 44].

2.3 Motivation

2.3.1 Debugging Concurrent PM Programs. Developing correct and efficient PM applications is hard. The low-level flush and fence instructions are flexible and efficient, but require the expertise in PM programming and are prone to cause consistency issues. Some high-level libraries, e.g., Persistent Memory Development Kit (PMDK) [10], provide simplified programming interfaces, e.g., transactions. Misunderstanding or misuse of library APIs also induces inconsistency or performance issues [44].

Existing testing tools for the crash consistency of hard-drive based file systems are inefficient for PM-based systems. Note that traditional hard-drive based file systems flush data at block granularity via only explicit software persistency operations, which are fundamentally different from the cache line granularity and hardware-based arbitrary cache eviction in PM [19, 28]. Such differences cause numerous states of data and significantly enlarge the search space of inputs and interleavings for PM programs.

It is important but challenging to identify crash-consistency bugs, especially for concurrent PM programs. Existing PM-based debugging tools record the PM accesses [35, 36] or track the persistency states [44] to detect the violation of consistency rules. However, these debugging tools fail to efficiently detect the bugs in concurrent PM programs due to the following two reasons: (1) existing PM-specific debugging tools do not consider the thread interleavings in concurrent programs, thus overlooking the inconsistency hidden in specific interleavings and (2) the input space of programs can be so huge that it is almost impossible to efficiently trigger bugs using exhaustive testing. We further present an example in concurrent PM applications and the consequences of inconsistency.

2.3.2 A Real-World Example. This example comes from a new bug PMRace found in P-CLHT [31], a concurrent chained hash

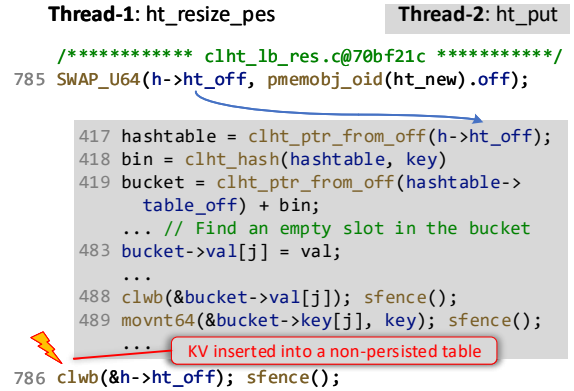


Figure 2: A PM Interleaving Concurrency Bug in P-CLHT found by PMRace.

index for PM. If the number of allocated buckets for chained linked lists exceeds a threshold, P-CLHT is resized by allocating a new hash table and migrating inserted key-value items to the new table. For concurrency control, P-CLHT leverages bucket-grained locks and the search operations are lock-free. While the hash table is resizing, threads performing write operations on buckets of the (old) hash table will be blocked or help migrate items until the resizing completes.

The concurrent execution of resizing and write operations in specific interleavings leads to potential data loss in P-CLHT. Figure 2 shows a simplified buggy execution of two threads. Specifically, thread-1 is resizing the table and swapping the global hash table pointer to a new hash table (line 785). Thread-2 reads the unflushed table pointer (line 417) – `h->ht_off` – and then inserts a key-value item into the new table (lines 483-489, `movnt64` denotes a non-temporal store). If a crash occurs before flushing the table pointer (line 786), the table is recovered to the old version in PM and the new item inserted by thread-2 is lost. Note that the acquisition of a bucket lock for thread-2's insertion is not blocked by resizing, since the bucket and its lock come from the new hash table.

For existing PM-specific debugging tools, it is hard to detect the consistency issue in Figure 2 due to the unawareness of reading non-persisted data hidden in interleavings. Some designs check if PM writes are persisted at the end of functions [44] or by manually annotated checkers [36] to avoid missing flushes or fences. However, for the execution of thread-1 in Figure 2, the update of table pointer is followed by `CLWB` and `SFENCE`, which would successfully pass the checking for missing persistency operations, causing false negatives. Some schemes [34, 35], including the testing tool in RECIPE [31], inject crashes before persistency instructions and check the crash consistency. However, these tools require expertise for the accurate injection of crashes. Moreover, even if the crash point is found, these tools do not explore interleavings and cannot detect the cross-thread crash-consistency issue hidden in the buggy interleaving shown in Figure 2. In summary, detecting such PM concurrency bugs requires interleaving exploration to find buggy interleavings and reports of real data inconsistencies (e.g., inserting key-value items based on the unflushed table pointer) to support bug diagnosis and fix. The observation of new PM concurrency bugs and requirements for bug detection motivate our PMRace.

3 PM CONCURRENCY BUG PATTERNS

This section provides the assumptions and definitions of our PM concurrency bug patterns (§3.1) and revisits the P-CLHT example based on the definitions (§3.2).

3.1 Assumptions and Definitions

We assume that a system crash renders all transient states of all threads/processes lost [22]. This failure model of PM is based on the assumption that CPU registers and caches are volatile, which is commonly used in real hardware (e.g., Intel Optane PM with asynchronous DRAM refresh (ADR) [23, 62]), computer systems [27, 31, 66, 69], and PM-specific testing tools [14, 18, 34, 44]. Enhancing CPU caches with durability is possible but requires hardware modifications [1, 67] or additional intel-specific extended ADR (eADR) support [49], which are not necessary for PM programming (refer to §6.6 for more discussions about eADR). Hence, in the context of this paper, following previous work [14, 18, 19, 34–36, 44], we assume that CPU caches are not included in the persistent domain.

We further provide complete definitions of the PM-specific concurrency bugs discussed in this work. We first define *PM Inter-thread Inconsistency Candidate*.

Definition 1. *A **PM Inter-thread Inconsistency Candidate** occurs, when one thread reads non-persisted data written by other threads.*

A *PM Inter-thread Inconsistency Candidate* occurs because, for store instructions, the visibility of data in cache does not guarantee the persistency in the PM. The mismatch between the points of visibility and persistency causes undetermined behaviors.

Note that *PM Inter-thread Inconsistency Candidates* can be bug-free. For example, if a system crashes just after a thread reads non-persisted data, the program will terminate immediately and the reading will not introduce any observable impacts after rebooting. Hence, we call cross-thread reading non-persisted data as a *candidate* for inconsistency. We use the definition of *PM Inter-thread Inconsistency* to refine this pattern.

Definition 2. *A **PM Inter-thread Inconsistency** is one PM Inter-thread Inconsistency Candidate that has durable side effects based on the non-persisted data.*

In Definition 2, *durable side effects* (side effects [55] + persistency) refer to actions durably changing program states, e.g., writing to PM (except the dependent non-persisted data), writing to disks, and sharing information with other programs. The inconsistency stems from the mismatch between the durable side effects and corresponding old dependent data in PM. Specifically, while the durable effects indicate the completion of previous operations, the dependent non-persisted write is lost upon a crash, thus causing data inconsistency. A verified *PM Inter-thread Inconsistency* (requiring programmer's efforts) is classified as a *PM Interleaving Concurrency Bug*.

PM Inter-thread Inconsistency Candidate and *PM Inter-thread Inconsistency* have their sequential variants. Specifically, we define a program execution in which one thread reads non-persisted data from its previous PM writes as a *PM Intra-thread Inconsistency Candidate*. If the thread further makes durable side effects based on non-persisted read data, we refer to such case as a *PM Intra-thread Inconsistency*.

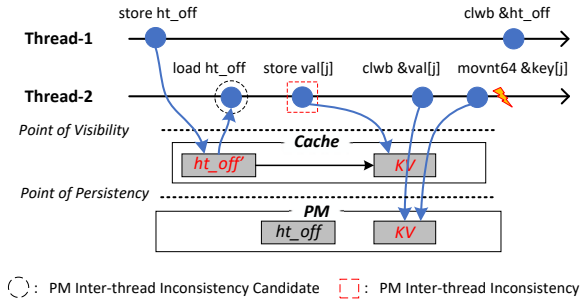


Figure 3: The timeline and data states of the P-CLHT example shown in Figure 2. (`SFENCE` instructions after `CLWB` and `movnt64` are not shown for clearness)

Definition 3. *A **PM Synchronization Inconsistency** occurs, when the synchronization data for concurrency control and the PM application's execution context are not in a consistent state after it restarts.*

If synchronization data such as locks and conditional variables are persisted to PM and restored after a crash, the execution context of the corresponding threads should be persisted and restored. Otherwise, if threads are reconstructed after restarts, the restored synchronization data will block the program execution. This pattern is also applied to the inconsistencies from exclusive resources, such as sockets. Similar to *PM Inter-thread Inconsistency*, the confirmed harmful *PM Synchronization Inconsistencies* belong to *PM Execution Context Bugs*.

3.2 Revisiting the P-CLHT Example

Based on previous definitions, we revisit the P-CLHT example. As shown in Figure 3, thread-2 reads the non-persisted table pointer stored by thread-1 in the CPU cache (i.e., ht_off'), thereby causing a *PM Inter-thread Inconsistency Candidate*. When thread-2 writes a new key-value item into PM based on the non-persisted table pointer, a *PM Inter-thread Inconsistency* occurs. A system failure after persisting the item and before the persistency of the new table pointer (i.e., ht_off') causes data loss: we cannot access the inserted item (i.e., KV) via the recovered table pointer (i.e., ht_off) in PM.

4 THE PMRACE DESIGN

PMRace is a PM-specific fuzzer to find PM concurrency bugs. To efficiently and accurately detect such bugs, there are three main problems: (1) *How to efficiently explore interleavings and find the buggy interleavings causing PM concurrency bugs?* (2) *For each fuzz campaign with an interleaving, how to accurately detect possible PM concurrency bugs?* (3) *How to reduce the false positives due to custom recovery mechanisms?*

We present the overview of PMRace (§4.1) and the main components in PMRace to address the above problems: PM-aware coverage-guided fuzzing to explore interleavings (§4.2), PM checkers to detect inconsistencies (§4.3), and a post-failure validation mechanism to reduce false positives (§4.4). We introduce PMRace's input generator for high-quality seeds (§4.5).

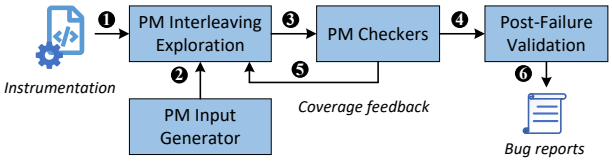


Figure 4: The PMRace overview.

4.1 Overview

PMRace is a PM-aware coverage-guided fuzzer to detect PM concurrency bugs. Figure 4 shows an overview of PMRace. Before the fuzz testing, the program needs to be instrumented by the LLVM [29] pass from PMRace (step ❶). By using the instrumented program and generated inputs from PMRace's operation mutator (step ❷), the interleaving exploration starts. PMRace leverages biased exploration strategies towards read-after-write PM accesses (step ❸). In each execution, our PM checkers detect the reading of non-persisted data (i.e., *PM Inter-thread Inconsistency Candidate*) and if there are any durable side effects based on the non-persisted data (i.e., *PM Inter-thread Inconsistency*). The changes of user-annotated synchronization variables in PM are recorded as *PM Synchronization Inconsistency* (step ❹). The results and coverage improvement from PM checkers are used as feedback to guide future interleaving exploration (step ❺). In the post-failure stage, PMRace verifies whether any durable side effects or modified synchronization variables are recovered to a consistent state. If so, the detected inconsistency is considered as a false positive; otherwise, PMRace generates a detailed bug report with stack traces and corresponding program inputs to facilitate bug diagnosis (step ❻).

4.2 PM-Aware Coverage-Guided Fuzzing

4.2.1 PM Alias Pair Coverage. Existing coverage metrics are unsuitable for the inconsistency detection. The reason is that conventional branch coverage (the coverage of explored code branches) [39, 58] and recent PM path (PM-specific branch coverage) [34] do not consider the interleavings of executions. Recent work proposes alias coverage [59], a metric for the coverage of instructions accessing the same memory address, for concurrent programs. However, alias coverage is insensitive to the persistency states of memory, thus being inefficient for crash-consistency bugs.

PMRace defines a new metric used for the coverage of PM-related interleavings, called *PM alias pair coverage* (or *PM alias coverage*). A PM access to address x is identified by (I_x, P_x, T_x), in which I_x, P_x, and T_x respectively denote the instruction ID (a unique integer assigned in PMRace's compiler pass), the persistency state of data, and the thread ID. A *PM alias instruction pair* (or *PM alias pair*) refers to two back-to-back PM accesses to the same address by different threads. For example, two back-to-back PM accesses are a PM alias pair, denoted by $\langle (I_x, P_x, T_x), (I_y, P_y, T_y) \rangle$, iff $x = y \wedge T_x \neq T_y$. PMRace maintains a bitmap in shared memory for the coverage of PM alias pairs (i.e., PM alias coverage).

4.2.2 PM-Aware Interleaving Exploration. Existing interleaving exploration strategies, e.g., enumerating thread interleavings [17, 24] and injecting delays at runtime [32, 46], do not consider the persistency of PM and become inefficient for the detection of PM

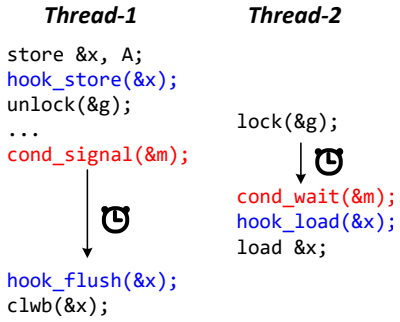


Figure 5: An example of injecting `cond_wait` and `cond_signal` to trigger inconsistency candidates in Figure 1 (Functions with "hook_*" indicate instrumented calls).

concurrency bugs. Unlike them, the PM-aware exploration strategy in PMRace selects and prioritizes the interleavings that may introduce inconsistency. The idea in the PM-aware interleaving exploration is to select some preemption points to drive programs towards reading non-persisted data.

There are three principles for the preemption point selection to debug a PM program: (1) Target PM accesses; (2) Focus on accesses to global PM data visible in the program, called *shared data accesses*; (3) Prioritize frequent shared data access instructions. The former two principles are straightforward, since we are interested in memory accesses that lead to possible inconsistencies. The third point stresses the checking on hot data in concurrent programs due to the high inconsistency possibility from frequent context switches. In general, the shared data are frequently accessed and often become the critical data (e.g., metadata) in the program, where the non-persistency tends to cause crash inconsistencies. In fact, the priority of preemption point is possible to be customized: different strategies for the priority can be implemented and integrated into PMRace's framework.

PMRace maintains a priority queue of PM shared data access instructions grouped by addresses. Each entry of the priority queue contains an address with corresponding load and store instructions. To explore a new interleaving, PMRace fetches one entry that has not been explored from the priority queue. Given one entry from the priority queue, PMRace tries to schedule the program execution towards reading non-persisted data, i.e., *PM Inter-thread Inconsistency Candidates*. The load instructions from the selected entry are called *sync points*. Specifically, PMRace inserts conditional waits (i.e., `cond_wait`) before these sync points; the signals of conditions (i.e., `cond_signal`) are triggered after corresponding store instructions from the selected entry but before the flush operations for the written data. The `cond_signal` will stall the writer thread for a while (a configurable parameter) to execute the load instructions in corresponding reader threads. Figure 5 presents an example of injecting `cond_wait` and `cond_signal` to trigger reading non-persisted `x` in thread-2.

The proposed thread scheduling is effective for simple cases, which however fails in a real-world system with complicated synchronization for concurrency control due to the following pitfalls.

Pitfall-1: The frequent execution of sync points causes a long testing time due to the stall for each sync point. In order to mitigate the overheads of conditional waiting, PMRace disables `cond_wait`

```
1  m = 0; // A condition variable
2
3  /* Before load instruction: wait for m */
4  void cond_wait(&m) {
5    if (sync.is_enabled && !t->bypass_sync) {
6      if (sync.skip == 0) {
7        while (!atomic_load(&m)) {
8          usleep(100);
9          if (/* Some threads block */) {
10           sync.is_enabled = 0;
11           break;
12         }
13         if (/* All threads block */) {
14           // Thread t is the privileged
15           t->bypass_sync = 1;
16           break;
17         }
18       }
19     }
20     else {
21       sync.skip--;
22     }
23   }
24 }
25
26 /* After store instruction: signal m */
27 void cond_signal(&m) {
28   atomic_store(&m, 1);
29   usleep(writerWaiting); // Wait for readers
30 }
```

Figure 6: The pseudo-code of the synchronization algorithm.

after receiving a signal. Specifically, as shown in Figure 6, the condition variable "m" is set to 1 in `cond_signal` (line 28), which will disable the while loop and `usleep` in `cond_wait` (lines 7-18) in current fuzz campaign, thus decreasing the testing time.

Pitfall-2: The system hangs because all threads are blocked (based on the number of executed while loops in lines 7-18) on sync points waiting for a signal from a writer thread that does not exist. A typical example is concurrent "ht_put" in P-CLHT for inserting key-value items: all threads are executing "ht_put" operations and waiting for the signal from the update of the table pointer. In such cases, as shown in Figure 6, PMRace randomly selects a thread as a privileged one (line 15), which receives the permission to bypass all `cond_wait` (line 5). It is possible that the randomly selected privileged thread does not execute corresponding store instructions and `cond_signal`. However, since we select the sync points from priority queue, it is of high possibility to encounter the corresponding store instructions in current fuzz campaign. Moreover, with enough fuzz campaigns, the random selection of privileged threads can cover the cases that the thread to execute `cond_signal` is selected. In the P-CLHT example, by using enough fuzz campaigns, it is expected that the selected privileged thread from four worker threads (see §6.1 for workload configurations) performs resizing, updates the table pointer, and sends a signal to other threads via `cond_signal`. Otherwise, stalled reader threads still wait for unavailable signals, which falls into the following case.

Pitfall-3: The system hangs because some threads are blocked on sync points. One reason for such hangs is that the privileged thread does not execute corresponding store instructions. Another reason is unnecessary blocking in the initialization or cleanup

stages. For such cases, when PMRace detects hangs in a thread due to `cond_wait`, it disables current sync point and stops waiting (line 10), as shown in Figure 6. At the end of this fuzz campaign, PMRace increases and saves the initial `skip` of all disabled sync points (`is_enabled = 0`), which indicates the execution times of `cond_wait` to be skipped in future (lines 6 and 21). Specifically, in the following fuzz campaigns using the same seed, PMRace loads and enables the saved sync point info, thus avoiding unnecessary blocking on the same sync point (lines 6 and 21).

The pseudo-code of the synchronization algorithm to trigger *PM Inter-thread Inconsistency Candidates* is presented in Figure 6. Hangs for some or all threads are handled in `cond_wait`. Note that the waiting time for store instructions (i.e., `writerWaiting`) depends on the execution of load instructions and durable side effects, which is supposed to be short. Hence, we set `writerWaiting` to the typical total execution time of the original program.

4.2.3 Coverage-Guided Fuzzing. PMRace leverages PM alias coverage and conventional branch coverage as the feedback to guide future fuzzing progress. If the coverages do not increase, PMRace tries the following three tiers of exploration.

- **Execution tier**: The program is executed multiple times before switching to another interleaving. For each execution, PMRace records coverages and updates the priority queue of shared data accesses. All detected *PM Inter-* and *Intra- thread Inconsistency (Candidates)* are saved.
- **Interleaving tier**: When repeated executions do not improve the coverages, we try another interleaving. For each interleaving, PMRace fetches an entry from the priority queue for sync points to detect inconsistencies.
- **Seed tier**: If fuzz campaigns with different executions and interleavings still do not increase the coverage, PMRace switches to another seed and reconstructs the priority queue.

4.3 PM Inconsistency Checkers

The runtime PM checkers in PMRace detect the inconsistency cases defined in §3.1. Specifically, PMRace instruments memory accesses (e.g., load/store instructions, non-temporal stores) and persistency instructions (e.g., `CLWB`) in PM programs. When encountering these instrumented instructions, corresponding hooked functions for checkers are invoked. We further present PMRace's checkers for PM concurrency bug patterns.

PM Inter-thread Inconsistency Candidate. In order to identify reading non-persisted data as inconsistency candidates (Definition 1), PMRace maintains a hash table to record the persistency states of PM data during runtime. Specifically, for a PM store instruction, PMRace finds its corresponding entry via the store address, updates the persistency state to `PM_DIRTY` (`PM_CLEAN` for non-temporal stores), and records current thread ID. The flush operations update the persistency states of corresponding regions to `PM_CLEAN`, indicating that previous PM writes are persisted. For a PM load instruction, if corresponding PM state is `PM_DIRTY`, an inconsistency candidate (i.e., reading non-persisted data) occurs. PMRace further checks the thread ID of previous store instruction for the address. If previous writer thread ID is different from current reader thread ID, it is a *PM Inter-thread Inconsistency Candidate*; otherwise, a *PM Intra-thread Inconsistency Candidate* occurs.

PM Inter-thread Inconsistency. To confirm if an inconsistency candidate causes crash-consistency issues, PMRace performs data flow analysis to detect durable side effects of reading non-persisted data. PMRace leverages dynamic taint analysis [45, 52] to check if there is data flow between previous inconsistency candidates and current PM writes. Specifically, there are two classes of data flows in PM writes that induce data inconsistencies: (1) *The contents to be written to PM are based on non-persisted data.* Such inconsistencies would cause unexpected data contents. (2) *The address of a PM store instruction is based on non-persisted data.* In this case, the data layout is inconsistent and may lead to data loss, e.g., the P-CLHT example in Figure 2. If one of the two types of data flow exists, PMRace confirms the durable side effects and reports a *PM Inter-thread (or Intra-thread) Inconsistency.*

PM Synchronization Inconsistency. Since thread contexts (e.g., registers and stack) are usually reconstructed, PMRace detects the updates of annotated synchronization variables in PM and marks each update as a *PM Synchronization Inconsistency*. Developers need to annotate synchronization data and their expected initial values via lightweight annotations (§5). Despite the numerous updates of PM synchronization data (e.g., locking/unlocking), PMRace checks each type of update operation for only one time, which significantly reduces the debugging overheads.

Implementing other PM checkers is possible by using PMRace's framework. For instance, unnecessary persistency operations can be detected by a checker in the hooked function for cache line flushes. Specifically, the checker needs to check if all data to be persisted exist in `PM_CLEAN` state via PMRace's hash table for persistency states. Another example is to check if PM data modified in a branch (e.g., basicblock in LLVM [29]) are persisted before the branch exits for the missing of flushes. We focus on PM concurrency bugs and leave other checkers for the future work.

4.4 Post-Failure Validation

Due to the existence of custom recovery mechanisms in some PM systems, some inconsistency issues are automatically fixed in the post-failure recovery stage and are benign. For example, memcached-pmem [12] rebuilds the LRU cache and the hash table from persistent slabs (i.e., persistent storage of inserted key-value items), which implicitly fixes some of the data inconsistencies found by PMRace and recent work [14] (e.g., inconsistencies limited to the "next" and "prev" fields of items are automatically fixed due to the index rebuilding), thus causing false positives. Another example is undo logging [10, 61] based transactions (e.g., PMDK), which copy old consistent data via write-ahead logging before transaction executions. During the recovery of uncommitted transactions, the data modified in transactions are reverted to the old consistent version from previous logs. Note that the undo logging based transactions in PMDK cannot avoid concurrency bugs. Unlike legacy transactions, PMDK's transactions do not guarantee the isolation for concurrent programs: the PM writes inside transactions are immediately visible to other threads [50].

Figure 7 shows an example of benign *PM Intra-thread Inconsistencies* detected by PMRace in clevel hashing [3], which is a lock-free PM hash index. During the construction of hash index, a new level object is allocated and assigned to a non-persisted level

```
/********** clevel_hash_ycsb.cpp@cae716f **********/
159 {
160   transaction::manual tx(pop);      Failure-atomic region
162   proot->cons = make_persistent<clevel_hash>();
      ...
165   transaction::commit();
166 }

/************* clevel_hash.hpp@cae716f *************/
278 clevel_hash() : meta(make_persistent<level_meta>()) {
      // Read non-persisted meta
294   m = convert_to_ptr(meta, my_pool_uuid);
      // Allocate a new level based on meta
300   m->first_level = make_persistent<level_bucket>();
      ...
320 }
```

Figure 7: A benign inconsistency example in clevel hashing.

pointer (line 300). However, this durable side effect of PM allocation does not lead to harmful impacts (e.g., PM leakage), since the outer PMDK transaction ensures the index construction is performed in an atomic manner (lines 160-165). If a crash occurs in this uncommitted transaction, the inconsistency will be fixed via rebuilding the index.

In order to reduce false positives due to application-specific recovery mechanisms, we propose a post-failure validation scheme to verify if detected inconsistencies are fixed in the recovery stage. Specifically, once an inconsistency is found in the pre-failure stage, PMRace duplicates the mmapped PM pool file at this crash point and records the address of durable side effects (*PM Inter-thread Inconsistency*) or the PM updates of synchronization variables (*PM Synchronization Inconsistency*). Note that the addresses are saved as offsets within the pool to avoid the effects of address space layout randomization (ASLR) [56]. PMRace leverages different mechanisms to automatically identify inconsistencies fixed in the recovery code (i.e., false positives).

(1) For each *PM Inter-thread Inconsistency*, the idea behind our automatic validation is to check if all inconsistent data are overwritten during the recovery stage. In the post-failure stage, PMRace restarts the tested program and mmaps the duplicated pool file. If all recorded inconsistent PM writes (durable side effects) are overwritten, PMRace marks the inconsistency as a false positive. This mechanism is also used for the validation of *PM Intra-thread Inconsistency*. The benign inconsistency shown in Figure 7 is automatic identified as a false positive via post-failure validation due to the overwriting of `m->first_level` during recovery.

(2) For each *PM Synchronization Inconsistency*, in the post-failure stage, PMRace checks if the detected synchronization variable update is correctly restored to the expected value, which comes from programmer's annotations (§5). If the synchronization variable is correctly reinitialized, corresponding synchronization inconsistency is benign.

PMRace's post-failure validation does not filter all false positives. The reason is that some PM programs leverage application-specific mechanisms, such as lazy recovery [21], checksums [12], and redo logging [51], to tolerate inconsistencies. For instance, many inconsistencies in FAST-FAIR B+-Tree [21] are lazily fixed upon future accesses to related keys. Hence, PMRace's validation during the immediate recovery stage misses these lazily recovered inconsistencies. For checksums and redo logging, the durable side effects occur in crash-consistent regions and are recovered by disregarding inconsistent contents with mismatched checksums or uncommitted redo logs. To address the missed false positive, PMRace provides a whitelist for developers to specify the benign reading of non-persisted data, e.g., crash-consistent reading protected by redo logging and checksums, by listing related locations of codes in the whitelist. When PMRace finds the stack trace of a detected inconsistency contains codes in the whitelist, PMRace marks the inconsistency to be safe and does not report this inconsistency as a bug. For example, the default whitelist of PMRace includes the transactional allocations in PMDK (redo logging) [51]. Hence, the PMDK awareness in the default whitelist transparently reduces the false positives for PMDK-based applications.

4.5 PM Input Generator

Unlike the default binary transformation based mutator [39] used in recent PM-specific fuzzer [34], our PMRace proposes an operation mutator to efficiently generate valid structured inputs that conform to the syntactic and semantic requirements of program interfaces. Our operation mutator originates from the fact that existing PM-based programs are usually in-memory applications with interactive APIs, e.g., key-value stores and indexes, which require structured inputs to be parsed by these programs. PMRace's input generator allows developers to provide operation rules (in the form of C code) based on existing examples. By efficiently exploiting these application-specific knowledge on inputs, PMRace generates high-quality seeds containing operation sequences for concurrent programs to efficiently achieve high coverage in "deeper" code (behind the input parsing stage).

Specifically, PMRace implements the custom mutations used by AFL++ [39]. Inspired by Krace [59], PMRace provides several evolution strategies to mutate existing seeds:

- **Mutation**: updating an arbitrary parameter of a random operation to another valid value.
- **Addition**: adding an operation at an arbitrary position.
- **Deletion**: deleting an arbitrary operation.
- **Shuffling**: shuffling operations and distributing to threads.
- **Merging**: merging two existing seeds into a new seed.

Different from Krace, PMRace prioritizes similar keys as operation parameters to increase the shared memory accesses and PM alias pairs. Moreover, if theses evolution strategies do not improve the branch coverage, our mutator tries to populate the PM systems by generating lots of "insert" operations with various keys. The load phase with many insertions effectively triggers resizing mechanisms in PM key-value stores and indexes. Based on these mutation strategies and awareness of PM system properties, PMRace efficiently generates high-quality seeds to improve the branch coverage.

To guarantee the speed of seed generation, the PM input generator is decoupled with the interleaving exploration so that the separate mutator runs fast without exploring interleavings. In the meantime, each round of execution in input generation and interleaving exploration begins with an empty PM pool to avoid the

side effects of previous PM pools. Reusing an old PM pool complicates the bug diagnosis, especially for concurrent programs. This problem is also studied in the "aging" OS problem for kernel bug detection [59, 60].

5 SYSTEM IMPLEMENTATIONS

We have implemented all components of PMRace. The compiler pass for instrumentation is based on LLVM-11 [29]. The PM checkers are implemented in a dynamic runtime library and the data flow analysis is based on the dynamic taint analysis from LLVM's DataFlowSanitizer [57]. We implement PMRace's operation mutator based on the AFL++ framework (v3.01) [39]. The main components of PMRace are connected by using Python scripts.

PM Awareness. In order to distinguish PM accesses from DRAM accesses, PMRace instruments the calls to memory-mapping related interfaces to obtain the mapped memory regions. If a memory-mapped pool path is on PM devices, the mapped memory is PM. In addition to the POSIX `mmap()`, PMRace also instruments the pool management APIs in PMDK. Hence, programs using raw `mmap()` or PMDK's API are both supported in PMRace.

Annotations for Synchronization Variables. An annotation interface, `pm_sync_var_hint(size, init_val)`, is provided in PMRace, in which "size" denotes the synchronization variable size and "init_val" denotes the expected reinitialized value after recovery. Essentially, this annotation interface is a wrapped macro based on Clang's `__attribute__((annotate()))` syntax. Since developers only need to annotate the definitions of synchronization variables or the fields in declarations, few annotations are required.

In-Memory Checkpoints for Input Generation. Due to the overheads of loading a PM pool file and pool initialization in PMDK's `libpmemobj`, each fuzz campaign spends lots of time to initialize the PM pool. PMRace leverages the `fork server` from AFL++ [39] with in-memory checkpoints for PM pools. After a PM pool is initialized, PMRace maintains only one in-memory copy of the pool and starts the `fork server`. Each forked fuzz campaign begins with a copy of the initialized pool, thus avoiding expensive initialization overheads. Note that these checkpoints should not be used for low-level PM libraries (e.g., `libpmem`) or mechanisms (§6.5).

Concurrent Fuzzing. PMRace supports concurrent fuzzing to accelerate the bug detection. The main fuzzing process starts several worker processes and dispatches seeds to these worker processes, thus enabling concurrent fuzz campaigns with low contention.

6 EVALUATION

6.1 Experimental Setup

Our experiments run on a 2-socket server with two Intel Xeon Gold 6230R CPUs. Each CPU has 26 cores and 52 threads. This system contains 1.5 TB Intel Optane Persistent Memory 100 Series (128 GB × 12 in *App Direct* mode [10]) and 192 GB DRAM (16 GB × 12). Our machine runs on Ubuntu 18.04 with Linux kernel version 5.4.0.

We have tested existing open-source concurrent PM systems based on PMDK (v1.9), including hashing-based indexes (P-CLHT from RECIPE [31], clevel hashing [3], CCEH [42]), a tree-based index (FAST-FAIR B+-Tree [21]), and a key-value store (memcached-pmem [12]). The details of the evaluated PM systems are shown in Table 1. Note that during our evaluation of RECIPE we only tested P-CLHT. The reason is that the implementations of other indexes in the open-source RECIPE project were based on "libvmmalloc" [11], in which the allocator did not ensure crash consistency [30]. Instead of the original level hashing [68] implemented on simulated PM, we tested an optimized PMDK-based scheme (i.e., clevel hashing). For each PM system, we have implemented a driver program to concurrently issue requests via the system interfaces. The number of threads in the driver program is set to 4, since 96% of traditional concurrency bugs on DRAM are guaranteed to manifest using 2 threads [37]. To accelerate the bug detection, PMRace runs concurrently using 13 worker processes by default (§5).

Table 1: The concurrent PM programs tested by PMRace.

Systems	Version	Scope	Concurrency
P-CLHT [31]	70bf21c	Static hashing	Lock-based
clevel hashing [3]	cae716f	PM-optimized hashing	Lock-free
CCEH [42]	46771e3	Extendible hashing	Lock-based
FAST-FAIR [21]	0f047e8	B+-Tree	Lock-based
memcached-pmem [12]	8f121f6	Key-value store	Lock-based

We summarize the bugs found by PMRace (§6.2), demonstrate the accuracy including the false positive reduction (§6.3), and quantitatively evaluate the PM-aware exploration strategies (§6.4) and the input generator in PMRace (§6.5). Due to the lack of debugging tools for PM concurrent bugs, we compare the following schemes to evaluate the interleaving exploration.

- **PMRace** is our proposed scheme to detect crash inconsistencies in concurrent PM programs.
- **Delay Inj** (Delay Injection) is implemented in PMRace's framework but using delay injection [32, 46, 59] for interleaving exploration. Before each PM access, we inject a random delay (1 millisecond at most) following a uniform distribution.

6.2 The Bugs Found by PMRace

As shown in Table 2, we have found 14 unique bugs in concurrent PM systems and 10 bugs are new. A *unique bug* is a group of bugs of reading non-persisted data written by the same store instruction or inconsistencies due to the same synchronization variable type. For PM concurrency bugs, PMRace has found 10 bugs (6 new bugs). All bugs in P-CLHT have been confirmed by authors. The details of bugs are as follows.

PM Concurrency Bugs (*PM Inter-thread Inconsistency* and *PM Synchronization Inconsistency*)**.** In reported *PM Inter-thread Inconsistencies*, we found 8 unique bugs, 4 of which are new bugs. For P-CLHT, the bug (Bug 1) is inserting items into a non-persisted new hash table (§2.3), causing item loss after restarts. Similar to P-CLHT, the bug in FAST-FAIR (Bug 8) comes from inserting data based on a non-persisted node pointer. For memcached-pmem, we have identified 6 bugs (Bugs 9-14). Specifically, the previous 2 new bugs (Bugs 9 and 10) write item values based on non-persisted item values, causing inconsistent data in memcached-pmem. The other 4 bugs (Bugs 11-14) are also reported by recent PMDebugger [14] as missing flush operations. Unlike PMDebugger, PMRace identified consequent durable side effects based on unflushed data, indicating the existence of crash inconsistencies. For instance, though the "prev" and "next" fields of items are rebuilt and tolerant to inconsistencies (§4.4), the following durable side effects (e.g., "slabs_clsid"

Table 2: The unique bugs found by PMRace.

"Inter": *PM Inter-thread Inconsistency* "Sync": *PM Synchronization Inconsistency*
"Intra": *PM Intra-thread Inconsistency* "Other": Bug 4 is an inconsistency candidate and Bug 5 is a DRAM concurrency bug

Systems	#	Type	New	Write code	Read code	Description	Impact
P-CLHT	1	Inter	✓	clht_lb_res.c:785	clht_lb_res.c:417	read unflushed table pointer and insert items	data loss
	2	Sync	✓	clht_lb_res.c:429		do not initialize bucket locks after restarts	hang
	3	Intra	✓	clht_lb_res.c:789	clht_gc.c:190	read unflushed table pointer and perform GC	PM leakage
	4	Other	✓	clht_lb_res.c:321	clht_lb_res.c:616	read unflushed keys	redundant PM writes
	5	Other	✓	clht_lb_res.c:526		do not release bucket locks in update	hang
CCEH	6	Sync	✓	CCEH.h:86		do not release segment locks after restarts	hang
	7	Intra	✓	CCEH.h:165	CCEH.cpp:171	read unflushed capacity and allocate segments	PM leakage
FAST-FAIR	8	Inter	✓	btree.h:560	btree.h:876	read unflushed pointer and insert data	data loss
memcached-pmem	9	Inter	✓	memcached.c:4292	memcached.c:2805	read unflushed value and write value	inconsistent data
	10	Inter	✓	memcached.c:4293	memcached.c:2805	read unflushed value and write value	inconsistent data
	11	Inter	✗	items.c:423	items.c:464	read unflushed "prev" and write "slabs_clsid"	inconsistent index
	12	Inter	✗	slabs.c:549	slabs.c:412	read unflushed "next" and write "it_flags" or value	inconsistent index
	13	Inter	✗	items.c:1096	memcached.c:2824	read unflushed "it_flags" and write value	inconsistent data
	14	Inter	✗	items.c:627	items.c:623	read unflushed "slabs_clsid" and write "slabs_clsid"	inconsistent index

Table 3: The results of PM concurrency bug detection.

"Inter-Cand": *PM Inter-thread Inconsistency Candidate* "Validated/Whitelisted FP": false positives identified by post-failure validation/whitelist

Systems	PM Interleaving Concurrency Bug					PM Execution Context Bug			
	Inter-Cand	Inter	Validated FP	Whitelisted FP	Bug	Annotation	Sync	Validated FP	Bug
P-CLHT	35	10	0	2	1	4	4	3	1
clevel hashing	6	2	0	2	0	0	0	0	0
CCEH	15	0	0	0	0	2	1	0	1
FAST-FAIR	179	69	3	2	1	0	0	0	0
memcached-pmem	266	79	62	0	6	0	0	0	0
Total	501	160	65	6	8	6	5	3	2

and "it_flags" fields) survive crashes and lead to possible inconsistent index (Bugs 11 and 12).

In reported *PM Synchronization Inconsistency*, we found a new bug in P-CLHT in which persistent bucket locks are not released after restarts (Bug 2). A similar bug about segment locks is found in CCEH (Bug 6). After crash recovery, these bugs cause hangs when accessing corresponding data due to never released locks.

PM Intra-Thread Inconsistency. We have found 2 new bugs due to *PM Intra-Thread Inconsistencies*. For P-CLHT, PMRace identifies the rehashing of items based on non-persisted "table_new" field (Bug 3). We further observe that the allocated memory for "table_new" leaks after failures. The bug in CCEH is allocating a segment array based on non-persisted "capacity" (Bug 7). After restarts, the "capacity" is undefined and the allocated segment array may be leaked. Note that PM leakage is more significant than conventional DRAM leakage, since the leaked PM cannot be automatically recycled by rebooting due to the non-volatility of PM [4, 13].

Other Bugs. Besides crash-consistency bugs, we have found one bug of redundant PM writes (Bug 4) and one conventional concurrency bug (Bug 5) in P-CLHT. During our testing, PMRace reported an inconsistency candidate of reading non-persisted buckets and we further found that the writing of buckets is unnecessary, which decreases the system performance. The concurrency bug comes from the missing of unlock in `clht_update()`, which leads to hang triggered during the fuzz testing.

6.3 False Positives

In Table 3, we present the results of PM concurrency bug detection including detected inconsistencies, filtered false positives, found unique bugs from inconsistencies, and required efforts for annotations. By checking the durable side effects, PMRace prunes 68.5% *PM Inter-thread Inconsistency Candidates*. The post-failure validation filters many false positives (automatically fixed cross-thread data or synchronization inconsistencies), especially for memcached-pmem (62 false positives). As a result, the number of remaining inconsistencies to be checked is often small, e.g., 17 inconsistencies for memcached-pmem. The only exception is FAST-FAIR, which tolerates inconsistencies by using lazy recovery mechanism. PMRace's default whitelist covers the transactional allocations in PMDK and checksum-based crash-consistent operations in memcached-pmem. Providing more rules in the whitelist about the application-specific crash-consistency guarantees in the tested programs will further decrease the false positive rate. Moreover, bugs can be quickly identified by programmers via PMRace's detailed reports with stack traces and corresponding seeds.

In terms of *PM Synchronization Inconsistency*, five inconsistencies are detected in P-CLHT and CCEH (other systems do not have persistent locks). Three benign cases are identified in the post-failure validation. Considering only 6 annotations are required in source codes of two PM concurrent programs, the efforts from programmers for annotations are slight and cost-efficient.

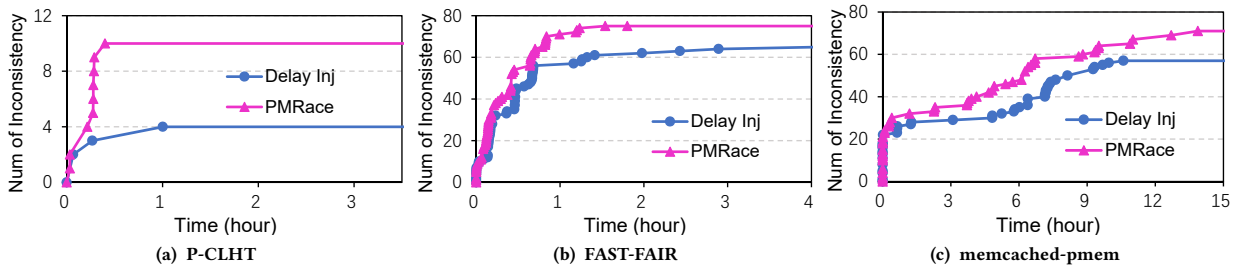
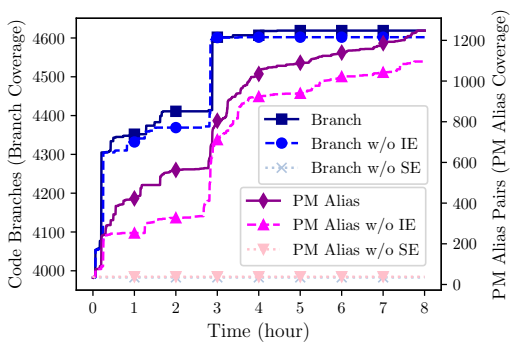


Figure 8: The time to identify *PM Inter-thread Inconsistency* (Each point indicates at least one inconsistency in an execution).



"IE": interleaving-tier exploration "SE": seed-tier exploration

Figure 9: The runtime-coverage of PMRace with P-CLHT.

Table 4: The code coverage of memcached-pmem commands.

"Error": invalid commands "Get*": get/bget commands
"Update*": add/set/replace/prepend/append commands

Schemes	Get*	Update*	incr	decr	delete	Error	Total
AFL++	154	647	132	143	148	707	2116
PMRace	190	902	179	183	166	0	2144

6.4 Exploration Efficiency

Figure 8 shows the time to find *PM Inter-thread Inconsistencies* in P-CLHT, FAST-FAIR, and memcached-pmem in the pre-failure stage (clevel hashing and CCEH are not included because no *PM Inter-leaving Concurrency Bug* was found). Due to the PM-aware thread scheduling, PMRace tries to block the PM read accesses until the write accesses to the same address occur, thus efficiently triggering reading non-persisted data (i.e., *PM Inter-thread Inconsistency Candidates*) than random delay injection at any PM accesses.

To measure the contribution of each exploration tier in PMRace, we conduct a case study of the runtime-coverage tradeoffs in P-CLHT using PMRace (single worker process) without interleaving-tier (w/o IE) or seed-tier exploration (w/o SE). As shown in Figure 9, PMRace w/o SE is difficult to improve the coverages, since one seed does not cover all possible executions. Unlike PMRace w/o IE, the interleaving-tier exploration in PMRace efficiently improves the coverages by searching and triggering inconsistency candidates. Conventional execution-tier exploration is known to be useful for coverage improvement for non-deterministic interleavings [32, 59]. In summary, all three exploration tiers are important to PMRace.

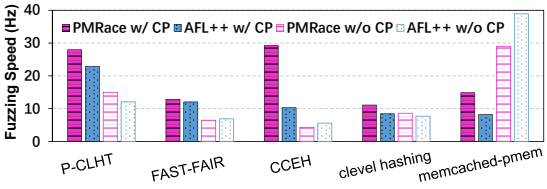


Figure 10: The impact of checkpoints (CP) in fuzzing.

6.5 Input Generator Efficiency

Since our operation mutator is based on AFL++ [39], we compare our mutator with the default mutator of AFL++.

The Improvement on PMRace's Mutator. AFL-COV [48] is used to measure the code coverage of random 100 seeds about input parsing, e.g., the `process_command()` in memcached-pmem, which is invoked about 2,100 times for each mutator. As shown in Table 4 (some valid commands are not presented), 1/3 of the commands in AFL++ are aborted due to invalid command inputs. Compared with the default mutator of AFL++, PMRace mutates seeds with semantic knowledge, thus generating high-quality seeds and satisfying the syntactic checking of program inputs to test "deeper" code and improve the code coverage.

The Impact of In-Memory Checkpoints. We measure the average fuzzing speed of input generators with or without in-memory checkpoints for pool initialization. As shown in Figure 10, for all tested workloads except memcached-pmem, both PMRace and AFL++ benefit from the in-memory checkpoints and respectively increase the fuzzing speeds by 199% and 65% on average, because the fuzz testing directly uses in-memory copies of initialized PM pools instead of re-constructing new ones. However, memcached-pmem leverages lightweight `pmem_map_file`, a simple wrapper of POSIX `mmap`, from `libpmem`, avoiding the expensive PM pool initialization in `libpmemobj`. Hence, we recommend disabling in-memory checkpoints when testing programs based on `libpmem`.

6.6 Discussions

Bug Coverage. Since our current implementation of PMRace focuses on PM concurrency bugs, PMRace does not detect sequential crash-consistency bugs (e.g., missing flush/fence operations) or traditional DRAM-based concurrency bugs in PM programs. A recent work [4] shows that traditional concurrency bugs still occur in PM programs and can cause persistent consequences — restarts do not solve the faults in PM systems. However, existing debugging

tools have proposed many detection mechanisms for PM sequential crash-consistency bugs [14, 18, 35, 44] or DRAM-based concurrency bugs [25, 32, 38, 54, 59], which are complementary to the cross-thread inconsistency checking in PMRace. To improve the bug coverage, it is possible to integrate existing checkers into PMRace's framework, e.g., adding existing PM checkers for flush/fence operations into PMRace's runtime library.

The Applicability of PMRace with eADR. When the eADR feature is available and enabled on a PM platform (requiring Intel Optane Persistent Memory 200 Series, the third generation Intel Xeon Scalable Processor, and OEM supports for additional batteries [49]), CPU caches become persistent domains protected by batteries. As a result, the cache line flush primitives are not required for PM programming [1, 49], thus avoiding *PM Inter-thread Inconsistency*. However, for *PM Synchronization Inconsistency* identified by PMRace, corresponding *PM Execution Context Bugs* still occur on eADR-based PM systems, since the unreleased locks and other exclusive resources in PM survive system crashes. After restarts, these unreleased resources will hinder the post-failure executions, e.g., uninitialized locks after recovery will block threads that need to acquire these locks.

7 RELATED WORK

Testing for PM Programs. There are two general classes of PM-related bugs studied by existing debugging tools: correctness bugs (e.g., inconsistent data) and performance bugs (e.g., missing or redundant flush/fence operations). Intel released Pmemcheck [6] and Persistency Inspector [7] to test PM programs by binary instrumentation, which, however, require accurate annotations. PMTest [36] detects the missing of flushes and persistency reordering by checking the violation of inserted assertions in memory access traces. Though the assertions from PMTest are flexible to use, the correct placements of assertions require the expertise in crash consistency. XFDetector [35] injects failure points and checks reading non-persisted data or semantically inconsistent data based on commit variables in the post-failure stage. PMFuzz [34] addresses the problem of test case generation in XFDetector by using PM path coverage with AFL++ [39]. The reading non-persisted pattern in XFDetector is similar to the *PM Inter-thread Inconsistency Candidate* of PMRace, but overlooks thread identity and only checks the violation in the post-failure stage. However, as discussed in §3.1, reading non-persisted may be not a bug, e.g., the benign cross-failure race defined in XFDetector. Different from XFDetector, our PMRace confirms the data inconsistency based on non-persisted data in the pre-failure stage. PMDebugger [14] optimizes data structures and summarizes nine rules of PM bugs to accelerate the debugging process. Unlike existing tools, AGAMOTTO [44] leverages symbolic execution with PM state machines to automatically find performance bugs. However, AGAMOTTO leaves correctness bugs to custom checkers, which requires the expertise in crash consistency and efforts to implement accurate checkers. Yat [28] is a testing tool for Intel's persistent memory file system (PMFS) [47]. The detection workflow in Yat enumerates the combinations of instructions for possible execution orders, which is cost-inefficient (5 years) for debugging [28]. Jaaru [19] leverages dynamic partial order reduction (DPOR) [16] to optimize the model checking overheads in Yat. Witcher [18] automatically detects correctness bugs for sequential PM programs by static analysis with output equivalence checking. Note that all above existing PM-specific debugging tools do not explore the possible thread interleavings in concurrent PM programs, thus leading to false negatives for *PM concurrency bugs*.

Testing for Concurrency Bugs. Concurrency bugs refer to bugs involving more than one thread. Common concurrency bugs can be classified into deadlock bugs and non-deadlock bugs [2]. Deadlocks occur when each thread holding resources (e.g., locks) are waiting for the resources held by other threads [26]. To avoid deadlocks, one solution is to acquire locks in a total order to prevent circular waiting [26]. Non-deadlock bugs include two major categories: atomicity-violation bugs and order-violation bugs [37]. Researchers have developed many tools to detect [25, 38, 54] or fix [63, 65] concurrency bugs based on the two patterns. However, concurrency bugs for PM are substantially different from conventional DRAM-based concurrency bugs due to persistency dimension and crash-consistency models (e.g., durable linearizability [22]). Some work focus on data race detection for concurrent programs [15, 46, 53, 59]. However, these tools for race detection also do not consider PM characteristics. PMRace focuses on PM programs and checks inconsistencies in read-after-write sequences.

Fuzzing for Bug Detection. Fuzz testing is widely used to find software bugs with different inputs (or thread interleavings). A common metric used in conventional fuzzers is branch coverage. During testing, fuzzers (e.g., AFL [64]) leverage the coverage (e.g., branch coverage) to guide the testing to use "new" inputs or interleavings, which improve the coverage, to find buggy executions [40]. PMFuzz [34] proposes PM path coverage to enhance the branch coverage with persistency awareness. However, both branch coverage and PM path coverage fail to represent the context changes of thread interleavings. In order to explore different thread interleavings, there are three classes of techniques: multiple runs with random scheduler, delay injection [32, 46, 59, 65], and interleaving enumeration [17, 24]. Unlike existing schemes, PMRace leverages the crash-consistency patterns in PM programs and focuses on the buggy read-after-write interleavings, thus significantly reducing the search space of interleavings.

8 CONCLUSION

For concurrent PM programs, crash-consistency bugs hidden in thread interleavings are challenging to detect. Existing PM-specific debugging tools do not explore interleavings in concurrent PM programs. In this paper, we identify two new types of PM concurrency bugs and propose PMRace to efficiently detect these concurrency bugs via PM-aware coverage-guided fuzzing. The false positives are reduced in the post-failure validation. In real-world concurrent PM systems, PMRace has found 10 new bugs.

ACKNOWLEDGMENTS

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and Key Laboratory of Information Storage System, Ministry of Education of China. We are grateful to our shepherd, Samira Khan, and the anonymous reviewers for their constructive comments and suggestions.

A ARTIFACT APPENDIX

A.1 Abstract

PMRace is a debugging tool to detect persistent memory (PM) concurrency bugs in PM systems. For hardware dependencies, the artifact needs 16 CPU threads and 32 GB DRAM. The minimal software requirements include vagrant and VirtualBox. The entry of artifact is a vagrant project to automatically construct the environments for the artifact evaluation on a VirtualBox virtual machine (VM). The evaluation spans the bug detection results in 5 PM systems, the time for *PM Inter-thread Inconsistency* detection, and the input generator efficiency.

A.2 Artifact check-list (meta-information)

- **Program:** PMRace
- **Compilation:** clang-11, clang++-11
- **Data set:** Open-source concurrent PM systems
- **Hardware:** 16 CPU threads and 32 GB DRAM
- **Run-time environment:** VirtualBox VM
- **Output:** Bug reports
- **Experiments:** Bug detection and performance evaluation
- **How much disk space required (approximately)?:** 100 GB
- **How much time is needed to prepare workflow (approximately)?:** 2 hours
- **How much time is needed to complete experiments (approximately)?:** 20 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GNU GPL v3.0
- **Workflow framework used?:** Vagrant
- **Archived (provide DOI)?:** TBD

A.3 Description

A.3.1 How to access. In addition to the archived version, we maintain a GitHub project: https://github.com/yhuacode/pmrace-vagrant.

A.3.2 Hardware dependencies. The artifact evaluation does not require real PM. In general, the VM needs 16 CPU threads and 32 GB DRAM. Therefore, the host machine needs to have enough hardware resources for the evaluation.

A.3.3 Software dependencies. Vagrant and VirtualBox need to be installed on a host machine running Linux.

A.3.4 Data sets. The artifact tests 5 open-source PM systems: P-CLHT in RECIPE, clevel hashing, CCEH, FAST-FAIR B+-tree, and memcached-pmem. Our scripts will automatically download and configure these workloads from GitHub.

A.4 Installation

After installing the software dependencies (Appendix A.3.3), the `vagrant up` command will complete the VM installation and setup in an automatic manner. In summary, the `vagrant up` command will automatically finish the following three main steps: (1) Construct a new VirtualBox VM using the configurations from the `Vagrantfile`; (2) Start the constructed VM and initialize the ssh settings; (3) Install the software prerequisites of the artifact in the VM (e.g., llvm-11).

Table 5: The number of unique bugs found by PMRace. ("n|m": n new bugs and m bugs in total).

Systems	Version	Inter	Sync	Intra	Other	Total
P-CLHT	70bf21c	1|1	1|1	1|1	2|2	5|5
clevel hashing	cae716f	-	-	-	-	-
CCEH	46771e3	-	1|1	1|1	-	2|2
FAST-FAIR	0f047e8	1|1	-	-	-	1|1
memcached-pmem	8f121f6	2|6	-	-	-	2|6
Total		4|8	2|2	2|2	2|2	10|14

Table 6: The numbers of detected inconsistencies and filtered false positives in the PM concurrency bug detection.

Systems	Inconsistencies (pre-failure)			False Positives (post-failure)		Bug
	Inter-Cand	Inter	Sync	Inter	Sync	
P-CLHT	35	10	4	0	3	2
clevel hashing	6	2	0	0	0	0
CCEH	15	0	1	0	0	1
FAST-FAIR	179	69	0	3	0	1
memcached-pmem	266	79	0	62	0	6
Total	501	160	5	65	3	10

A.5 Experiment workflow

The goal of PMRace's fuzzing is to identify durable side effects based on non-persisted data (i.e., *PM Inter- and Intra- thread Inconsistency*) and unreleased synchronization data (i.e., *PM Synchronization Inconsistency*). To debug a PM program, PMRace first instruments the source code via its LLVM pass. During the executions, inconsistencies are detected by PM checkers, which are implemented in the runtime library. The coverage of concurrent accesses to shared PM data (i.e., PM alias coverage) is leveraged as feedback to guide the interleaving exploration of fuzzing progress. In the post-failure stage, PMRace detects the automatically fixed durable side effects and synchronization data during recovery (i.e., false positives). For each detected inconsistency, PMRace generates a detailed bug report.

A.6 Evaluation and expected results

The experiments are classified into two categories: bug detection and performance evaluation. The experimental evaluation is automated using the provided scripts in the artifact and covers the following key results:

(1) The unique bugs found in 5 PM systems (Table 5, a summarized version of Table 2)
(2) The inconsistencies and false positives in the PM concurrency bug detection (Table 6, a summarized version of Table 3)
(3) The time to detect *PM Inter-thread Inconsistency* (Figure 8)
(4) The code coverage of memcached-pmem commands (Table 4)
(5) The impact of checkpoints for input generation (Figure 10)

After the installation of VM, users can log in to the VM via the `vagrant ssh` command. The details about running experiments are presented in the "EXPERIMENTS.md" of the artifact.

Note that due to the differences in hardware environments and fuzzing time, the evaluation results and inconsistencies may be not exactly the same with those reported in the paper. Increasing the fuzzing time and the scale of seeds are beneficial to the PM bug detection.

A.7 Experiment customization

Though the experiments are run in the VM, the artifact can be installed on a bare-metal machine. Our scripts, e.g., "bootstrap.sh", "setup.sh", and "build_*.sh" in "pmrace-vagrant", contain the steps to install all software dependencies for the artifact (tested on Ubuntu 18.04).

REFERENCES

[1] Mohammad A. Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)*. Seoul, South Korea. https://doi.org/10.1109/HPCA51647.2021.00019

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces* (1.00 ed.). Arpaci-Dusseau Books.

[3] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*. https://www.usenix.org/conference/atc20/presentation/chen

[4] Brian Choi, Randal Burns, and Peng Huang. 2021. Understanding and Dealing with Hard Faults in Persistent Memory Systems. In *Sixteenth European Conference on Computer Systems (EuroSys '21)*. Online Event, United Kingdom. https://doi.org/10.1145/3447786.3456252

[5] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. 2020. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface. In *2020 USENIX Annual Technical Conference (USENIX ATC '20)*. https://www.usenix.org/conference/atc20/presentation/choi

[6] Intel Corporation. 2018. 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.

[7] Intel Corporation. 2018. 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.

[8] Intel Corporation. 2019. Intel Optane DC persistent memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[9] Intel Corporation. 2020. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html.

[10] Intel Corporation. 2020. Persistent Memory Development Kit. http://pmem.io/.

[11] Intel Corporation. 2021. The libvmmalloc man page. https://pmem.io/pmdk/manpages/linux/v1.3/libvmmalloc.3.html.

[12] Lenovo Corporation. 2018. Memcached-pmem. https://github.com/lenovo/memcached-pmem.

[13] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC '18)*. Boston, MA, USA. https://www.usenix.org/conference/atc18/presentation/david

[14] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Virtual Event, USA. https://doi.org/10.1145/3445814.3446744

[15] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Dublin, Ireland. https://doi.org/10.1145/1542476.1542490

[16] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. Long Beach, California, USA. https://doi.org/10.1145/1040305.1040315

[17] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI '14)*. Broomfield, CO, USA. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/fonseca

[18] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Virtual Event. https://doi.org/10.1145/3477132.3483556

[19] 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)*. New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446735

[20] Frank T. Hady, Annie P. Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017), 1822–1833. https://doi.org/10.1109/JPROC.2017.2731776

[21] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST '18)*. Oakland, CA, USA. https://www.usenix.org/conference/fast18/presentation/hwang

[22] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *30th International Symposium on Distributed Computing (DISC '16)*, Vol. 9888. Paris, France. https://doi.org/10.1007/978-3-662-53426-7_23

[23] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[24] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (S&P '19)*. San Francisco, CA, USA. https://doi.org/10.1109/SP.2019.00017

[25] Guoliang Jin, Aditya V. Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '10)*. Reno/Tahoe, Nevada, USA. https://doi.org/10.1145/1869459.1869481

[26] Edward G. Coffman Jr., M. J. Elphick, and Arie Shoshani. 1971. System Deadlocks. *ACM Comput. Surv.* 3, 2 (1971), 67–78. https://doi.org/10.1145/356586.356588

[27] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Huntsville, ON, Canada. https://doi.org/10.1145/3341301.3359631

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

[29] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO '04)*. San Jose, CA, USA. https://doi.org/10.1109/CGO.2004.1281665

[30] Se Kwon Lee. 2020. The RECIPE Project. https://github.com/utsaslab/RECIPE/tree/70bf21c6240327f4f8fac343aba708f194fe19f4.

[31] 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)*. Huntsville, ON, Canada. https://doi.org/10.1145/3341301.3359635

[32] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-SafetyViolation Detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Huntsville, ON, Canada. https://doi.org/10.1145/3341301.3359638

[33] ARM Limited. 2020. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/ga.

[34] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. PMFuzz: Test Case Generation for Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. New York, NY, USA, 16 pages. https://doi.org/10.1145/3445814.3446691

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

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

[37] 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)*. Seattle, WA, USA. https://doi.org/10.1145/1346281.1346323

[38] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*. San Jose, CA, USA. https://doi.org/10.1145/1168857.1168864

[39] Dominik Maier, Heiko Eißfeldt, Andrea Fioraldi, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT '20)*. https://www.usenix.org/conference/woot20/presentation/fioraldi

[40] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Software Eng.* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[41] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

[42] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*. Boston, MA, USA. https://www.usenix.org/conference/fast19/presentation/nam

[43] Ian Neal, Andrew Quinn, and Baris Kasikci. 2021. Hippocrates: Healing Persistent Memory Bugs without Doing Any Harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. New York, NY, USA, 14 pages. https://doi.org/10.1145/3445814.3446694

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

[45] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '05)*. San Diego, California, USA. https://www.ndss-symposium.org/ndss2005/dynamic-taint-analysis-automatic-detection-analysis-and-signaturegeneration-exploits-commodity/

[46] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. Washington, DC, USA. https://doi.org/10.1145/1508244.1508249

[47] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Ninth Eurosys Conference 2014 (EuroSys '14)*. Amsterdam, The Netherlands. https://doi.org/10.1145/2592798.2592814

[48] Michael Rash. 2018. AFL-COV. https://github.com/mrash/afl-cov.

[49] Steve Scargall. 2020. *Programming Persistent Memory: A Comprehensive Guide for Developers.* Apress, Berkeley, CA, Chapter Persistent Memory Architecture, 11–30. https://doi.org/10.1007/978-1-4842-4932-1_2

[50] Steve Scargall. 2020. *Programming Persistent Memory: A Comprehensive Guide for Developers.* Apress, Berkeley, CA, Chapter Concurrency and Persistent Memory, 277–294. https://doi.org/10.1007/978-1-4842-4932-1_14

[51] Steve Scargall. 2020. *Programming Persistent Memory: A Comprehensive Guide for Developers.* Apress, Berkeley, CA, Chapter PMDK Internals: Important Algorithms and Data Structures, 313–331. https://doi.org/10.1007/978-1-4842-4932-1_16

[52] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy (S&P '10)*. Berleley/Oakland, California, USA. https://doi.org/10.1109/SP.2010.26

[53] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the workshop on binary instrumentation and applications*.

[54] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I Use the Wrong Definition? DefUse: Definition-Use Invariants for Detecting Concurrency and Sequential Bugs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '10)*. Reno/Tahoe, Nevada, USA. https://doi.org/10.1145/1869459.1869474

[55] David A. Spuler and A. Sayed Muhammed Sajeev. 1994. Compiler Detection of Function Call Side Effects. *Informatica (Slovenia)* 18, 2 (1994).

[56] PaX Team. 2003. PaX Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt.

[57] The Clang Team. 2020. DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html.

[58] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '19)*. Chaoyang District, Beijing, China. https://www.usenix.org/conference/raid2019/presentation/wang

[59] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (S&P '20)*. San Francisco, CA, USA. https://doi.org/10.1109/SP40000.2020.00078

[60] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *2019 IEEE Symposium on Security and Privacy, (S&P '19)*. San Francisco, CA, USA. https://doi.org/10.1109/SP.2019.00035

[61] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-execute More. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Virtual Event, USA. https://doi.org/10.1145/3445814.3446730

[62] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*. Santa Clara, CA, USA. https://www.usenix.org/conference/fast20/presentation/yang

[63] Jie Yu and Satish Narayanasamy. 2009. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *36th International Symposium on Computer Architecture (ISCA '09)*, Stephen W. Keckler and Luiz André Barroso (Eds.). Austin, TX, USA. https://doi.org/10.1145/1555754.1555796

[64] Michal Zalewski. 2017. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[65] Mingxing Zhang, Yongwei Wu, Shan Lu, Shanxiang Qi, Jinglei Ren, and Weimin Zheng. 2014. AI: A Lightweight System for Tolerating Concurrency Bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. https://doi.org/10.1145/2635868.2635885

[66] Wen Zhang, Scott Shenker, and Irene Zhang. 2020. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. https://www.usenix.org/conference/osdi20/presentation/zhang-wen

[67] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support. In *The 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '13)*. Davis, CA, USA. https://doi.org/10.1145/2540708.2540744

[68] Pengfei Zuo. 2018. The Level Hashing Project. https://github.com/Pfzuo/Level-Hashing.

[69] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. Carlsbad, CA, USA. https://www.usenix.org/conference/osdi18/presentation/zuo