PMFuzz: Test Case Generation for Persistent Memory Programs

Sihang Liu*
University of Virginia
Charlottesville, Virginia, USA
sihangliu@virginia.edu

Suyash Mahar*
University of California, San Diego
San Diego, California, USA
smahar@ucsd.edu

Baishakhi Ray
Columbia University
New York City, New York, USA
rayb@cs.columbia.edu

Samira Khan
University of Virginia
Charlottesville, Virginia, USA
samirakhan@virginia.edu

ABSTRACT

The Persistent Memory (PM) technology combines the persistence of storage with the performance approaching that of DRAM. Programs taking advantage of PM must ensure data remains recoverable after a failure (e.g., power outage), and therefore, are susceptible to having crash consistency bugs that lead to incorrect recovery after a failure. Prior works have provided tools, such as Pmemcheck, PMTest, and XFDetector, that detect these bugs by checking whether the trace of PM accesses violates the program's crash consistency guarantees. However, detection of crash consistency bugs highly depends on test cases—a bug can only be detected if the buggy program path has been executed. Therefore, using a test case generator is necessary to effectively detect crash consistency bugs.

Fuzzing is a common test case generation approach that requires minimum knowledge about the program. We identify that PM programs have special requirements for fuzzing. First, a PM program maintains a persistent state on PM images. Therefore, the fuzzer needs to efficiently generate valid images as part of the test case. Second, these PM images can also be a result of a previous crash, which requires the fuzzer to generate crash images as well. Finally, PM programs can have various procedures but only those performing PM operations can lead to crash consistency issues. Thus, an efficient fuzzer should target those relevant regions. In this work, we provide PMFuzz, a test case generator for PM programs that meets these new requirements. Our evaluation shows that PMFuzz covers 4.6× more PM-related paths compared to AFL++, a widely-used fuzzer. Further, test cases generated by PMFuzz discovered 12 new real-world bugs in PM programs which have already been extensively tested by prior PM testing works.

CCS CONCEPTS

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

KEYWORDS

Persistent Memory, Crash Consistency, Testing, Debugging, Fuzzing

ACM Reference Format:
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), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3445814.3446691

1 INTRODUCTION

Persistent memory (PM) technologies, such as Intel's Optane [30], provide a class of high-performance and byte-addressable memory. The use of PM allows a program to directly access persistent data through the load/store interface, without using software intermediaries. Thus, it blurs the boundary between memory and storage. As Intel's PM modules become widely available on the market [30] and are getting deployed in data centers [5, 6], a myriad of real-world applications have been developed for PM, such as databases [29, 39, 51], key-value stores [4, 31, 84, 85], customized PM applications [3, 12, 13, 16, 28, 67, 78, 89], and PM libraries that improve programmability [15, 26, 32, 79]. These software systems generally require that the persistent data can recover to a consistent state in the event of a failure (e.g., a power outage or system crash)—a requirement referred to as the crash consistency guarantee.

However, due to the reordering and buffering in the volatile memory hierarchy, writes to PM need to be carefully managed to ensure crash consistency. For example, appending a node to a persistent linked list requires the node to become persisted *prior* to the updated tail pointer that points to the new node. To prescribe the order in which writes become persistent, PM hardware systems have introduced new instructions, such as `CLWB` and `SFENCE` from x86 [38]. With the hardware support, programming for PM systems becomes possible but remains challenging—programmers need to have a good knowledge of both their programs and the hardware primitives. PM libraries, such as Intel's PMDK [32], improve the programmability by providing a higher-level interface. However, programmers still need to understand the crash consistency guarantees from the library and the desired failure-recovery mechanism

[*] Equal contribution. Suyash Mahar contributed to this work during his internship at the University of Virginia.

ASPLOS '21, April 19–23, 2021, Virtual, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446691

in their programs. Prior works have pointed out that programming for PM systems is error-prone [10, 49, 57, 58, 71]. A *misuse* of PM primitives or library functions, such as missing CLWB and SFENCE operations or not backing up data, can break the crash consistency guarantees, which is referred to as a *crash consistency bug*. Whereas, *overuse* of these functions, such as placing a redundant SFENCE or making unnecessary backups, can degrade the performance, which is referred to as a *performance bug*.

To mitigate the difficulties in PM programming, there have been testing tools that detect crash consistency bugs, as well as performance bugs [10, 49, 57, 58, 66], by tracing PM operations and determining whether they violate any of the crash consistency guarantees. However, a major issue remains unsolved—these testing tools still require the *buggy procedure to be actually executed*. For example, to reproduce a bug in PMDK [37] that was reported by PMTest [58], the inputs to a B-Tree-based key-value store need to be carefully designed, in order to execute a program path that triggers B-Tree's insertion and rebalancing procedures. Hence, even with the aid of PM testing tools, bugs cannot be detected without having inputs to trigger the required execution path. In this work, we aim to assist PM programming by *generating test cases* to cover nontrivial crash consistency and performance bugs.

Due to the already complicated programming for PM systems, a tool for test case generation ideally should not place an additional burden on programmers. *Fuzzing*, a widely-used test case generation method, perfectly satisfies this demand as it requires minimum knowledge about the target code base and has been proven to be effective [8, 18, 20, 24, 91]. At a high-level, a fuzzer *iteratively* generates new test cases by mutating existing ones, where high-value test cases, such as those that explore new branches, are reused in future iterations. Although fuzzing is an effective method, we identify that in order to generate test cases for PM programs efficiently, additional requirements need to be satisfied.

First, PM programs maintain the persistent state on PM devices (e.g., as a PM image in a DAX file system), different from conventional programs. A PM program takes not only the *regular program input* (e.g., a command that inserts a key-value pair) but also a *PM image* which contains an existing persistence state. As the procedure of loading an existing PM image and performing operations on top can also face crash consistency bugs [49, 57], it is necessary for a fuzzer to provide PM images as inputs. Fuzzers for conventional programs perform mutation to generate regular inputs (e.g., commands). In comparison, PM images have a much larger exploration space (e.g., tens of MBs). Therefore, generating PM images through direct mutation is ineffective and will likely produce invalid images. For example, a randomly mutated PM image may have illegal pointers that may cause the program to abort in the beginning without exploring any useful paths. Even though recent works have designed fuzzers for file system images, they require a well-defined image layout [44, 88]. As PM programs tend to customize the persistent data management, methods taken by file system fuzzers are not suitable for PM image generation. Therefore, the *first challenge* is to efficiently generate *valid PM images*.

Second, PM programs also need to recover from PM images that are resulted from failures during program execution, which we refer to as *crash images*. Prior works have shown that the recovery procedure is also susceptible to crash consistency bugs [49, 57]. Therefore, the fuzzer needs to generate not only *normal PM images* but also *crash images* for thorough testing. However, a program can fail at any point during execution, leading to a potentially infinite number of crash images. Therefore, the *second challenge* is to generate *crash images* that are most effective for testing.

Finally, PM programs may contain procedures for different purposes, not limited to managing PM, especially in real-world workloads. On the other hand, only PM operations are critical to crash consistency bugs—performing writes to PM without taking care of their ordering can leave inconsistent data on PM, and reading from them can cause the later execution to behave incorrectly [57]. However, traditional coverage metrics, such as branch coverage, used by conventional fuzzers do not target procedures with the most concerned PM operations. Therefore, the *third challenge* is to design a fuzzer that can *target PM-related procedures*.

The new requirements for test case generation are critical to systematically testing PM programs. However, existing fuzzers are incapable of meeting these requirements. In this work, we develop PMFuzz (available at https://pmfuzz.persistentmemory.org), a fuzzer that aims to generate test cases for detecting crash consistency and performance bugs in PM programs. Next, we describe the three high-level ideas of our design.

PM Image Generation. Existing fuzzers either do not target large PM images or require a fixed image layout, as directly mutating an image can likely generate invalid images that cannot explore useful paths. Therefore, an effective image generation method should guarantee valid PM images. We observe that a PM image is essentially an outcome of input commands. Therefore, our key idea is to leverage the program logic to *mutate* an existing PM image. PMFuzz incrementally generates the image by applying the fuzzing logic on the input commands. And eventually, the PM image will be thoroughly mutated through the iterative fuzzing procedure.

Crash Image Generation. In addition to taking *normal images* as inputs, PM programs can also execute on *crash images* that are caused by failures. Although a failure can occur at any point during execution, the recovery procedure typically depends on a few key variables that are stored in the image. For example, an undo-log-based program performs the following steps: back up the old data in the undo log, set the valid bit of the log, perform in-place update, and finally unset the valid bit. In case of a failure, the recovery procedure will take one of these two paths depending on the value of the valid bit: one path applies the undo log and the other directly resumes the execution. As such, there is a *control-flow dependency* between the execution before and after the failure. Based on this dependency, only two failure images are needed to cover both paths: one with the valid bit set to one and another set to zero. Our key idea is to minimize the number of crash images by only generating the images that can affect the control-flow in the recovery procedure.

Coverage of PM Path. As crash consistency and performance bugs are caused by the misuse of PM operations, achieving high coverage of these bugs requires the fuzzer to perform a *targeted fuzzing* on program paths with PM operations. To enable this prioritization, we first define the *PM path* as a path that consists of program statements with PM operations (e.g., read, write, writeback, etc.). Then, PMFuzz monitors the statistics of PM paths during fuzzing, and prioritizes test cases that cover *new* PM paths. By focusing on PM paths, PMFuzz can efficiently generate more test cases that target crash consistency and performance bugs.

Based on the key insights above, we implement PMFuzz on top of an open-source fuzzer, AFL++ [20], and evaluate it in a real PM system. Our contributions are the following:

- PMFuzz is the first test case generator for detecting crash consistency and performance bugs in PM programs.
- We evaluate PMFuzz using eight representative PM programs in a real PM system. On average, PMFuzz covers 4.6× more PM paths over the well-known fuzzer, AFL++, within 4 hours of fuzzing.
- Even though these PM programs have been extensively tested by prior works [10, 57, 58, 66], we detect 12 new real-world bugs with PMFuzz's systematic test case generation.

2 BACKGROUND AND MOTIVATION

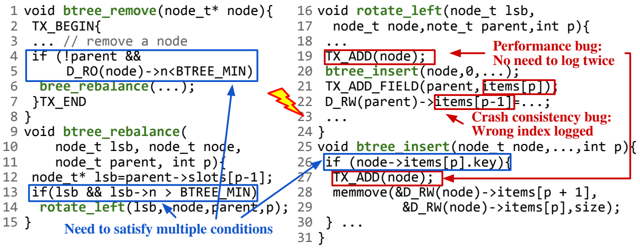
In this section, we first introduce the PM programming interface that guarantees recoverability and then describe the difficulties.

2.1 Programming for PM Systems

Persistent memory (PM) technologies, such as Intel's Optane [30], provide high-speed and byte-addressable access to persistent data. Programs can better leverage the performance of PM by directly managing persistent data in PM and bypassing the OS indirections (e.g., file systems). A common approach is to create a PM image in a file system with the direct access support (e.g., Ext4-DAX), map it to the program's address space, and manipulate the persistent data [77]. Recent PM applications, such as databases and key-value stores [4, 29, 31, 39, 51, 84, 85], PM-optimized file systems [19, 42, 48, 86, 87], PM libraries [15, 26, 32, 79], and other customized applications that are built upon those libraries [3, 12, 13, 16, 28, 67, 89] directly manipulate memory to avoid the OS overhead.

Programs developed for PM typically require data to be recoverable in case of a failure, which we refer to as the *crash consistency guarantee*. However, due to the reordering and buffering in the memory hierarchy, the order a write becomes persistent may differ from what the program intends to. To support programming for PM systems, hardware platforms have introduced new instructions. For example, in an x86 system, a sequence of "CLWB;SFENCE" instructions [38] ensures that a cache line will be persisted prior to subsequent writes (usually referred to as a persist_barrier()); in an ARM system, similar functionalities can be implemented using a sequence of "DC CVAP;DSB" instructions [2]. Building upon these primitives, PM libraries provide software interfaces, such as transactions [15, 26, 32, 79] and persistent data structures [16, 78], for better programmability. For example, Intel's PMDK library [32] provides a transaction interface, with wrappers such as TX_BEGIN and TX_END that mark failure-recovery regions, TX_ADD() that performs logging, and D_RO and D_RW (direct read-only/read-write) that obtain pointers to objects in the memory-mapped PM image.

These programming interfaces make it easier to manage persistent data and develop crash consistency mechanisms, such as undo/redo logging [14, 25, 28, 32, 46, 86], shadow paging [27, 53, 65], and checkpointing [21, 43, 72, 82]. However, it is not easy to implement such mechanisms—programmers need to have good knowledge about both the requirements for recovery and the persistence guarantees of PM programming support. Next, we will use an example to illustrate non-trivial bugs in PM programming.



```
 1 void btree_remove(node_t* node){
 2  TX_BEGIN{
 3  ... // remove a node
 4  if (!parent &&
 5       D_RO(node)->n<BTREE_MIN)
 6   bree_rebalance(...);
 7  }TX_END
 8 }
 9 void btree_rebalance(
10     node_t lsb, node_t node,
11     node_t parent, int p){
12  node_t* lsb=parent->slots[p-1];
13  if(lsb && lsb->n > BTREE_MIN)
14   rotate_left(lsb, node,parent,p);
15 }
```

```
16 void rotate_left(node_t lsb,
17   node_t node,note_t parent,int p){
18  ...
19  TX_ADD(node);
20  btree_insert(node,0,...);
21  TX_ADD_FIELD(parent,items[p]);
22  D_RW(parent)->items[p-1]=...;
23  ...
24 }
25 void btree_insert(node_t node,...,int p){
26  if (node->items[p].key){
27   TX_ADD(node);
28   memmove(&D_RW(node)->items[p + 1],
29           &D_RW(node)->items[p],size);
30  } ...
31 }
```

Performance bug: No need to log twice

Crash consistency bug: Wrong index logged

Need to satisfy multiple conditions

Figure 1: A buggy PM-based B-Tree (Example 1).

2.2 Nontrivial Bugs in PM Programming

Example 1: A buggy B-Tree. Figure 1 (Example 1) shows a simplified code snippet of a B-Tree that is implemented with PMDK's transaction library. The btree_remove() and btree_insert() procedures are wrapped inside a pair of TX_BEGIN and TX_END to ensure a consistent recovery after failure. Within the procedure, TX_ADD() is used to make a backup of the persistent data before it is modified. B-Tree is a commonly-used structure for key-value stores, where each node contains a number of keys. To remove an existing key from a B-Tree, the program first calls btree_remove(). After removal, if the number of keys (n) becomes less than BTREE_MIN, it rebalances the tree by calling btree_rebalance() (line 4-6), which left-rotates the modified node if the number of keys in its left sibling (lsb) exceeds BTREE_MIN (line 13-14). During the rotation process, rotate_left() calls the insertion function btree_insert() (line 18), which then checks the validity of the key (line 23), and performs the rotation (line 28-29). Finally, after insertion, rotate_left() updates items in its parent node (line 21-22).

Although this example seems to be correct as the whole procedure is wrapped in a transaction, there are two bugs. The first one is a crash consistency bug, where the program updates the (p-1)-th item (line 22) but logs the p-th item by mistake (line 21). In case of a failure at line 22, the item being modified can be lost as it has not been backed up by the log. The second one is a performance bug, where rotate_left() and btree_insert() attempt to log the same node twice (line 19 and 27), leading to unnecessary performance degradation.

These bugs in Example 1 have one major similarity that is they cannot be directly observed by programmers. A crash consistency bug, such as incorrect ordering or backup, does to affect the current volatile state, thus is not visible until a failure occurs during the buggy procedure. And, a performance bug, such as using excessive ordering or unnecessary logging, does not affect the ongoing execution. To make these bugs visible to programmers, there have been tools tailored for PM programming [10, 57, 58, 66]. These tools keep track of PM operations at runtime, and then detect violations against the crash consistency guarantees. These tools have the capability of detecting the bugs in Example 1. Nonetheless, they all *require the buggy program path to be executed* in order to detect the violations. In Example 1, the program needs to satisfy two if conditions to detect the crash consistency bug (line 21-22). Even harder, triggering the performance bug (line 27) requires satisfying all three

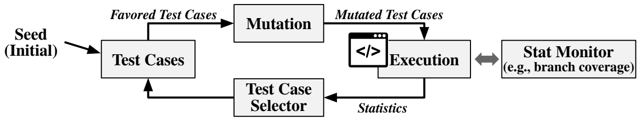
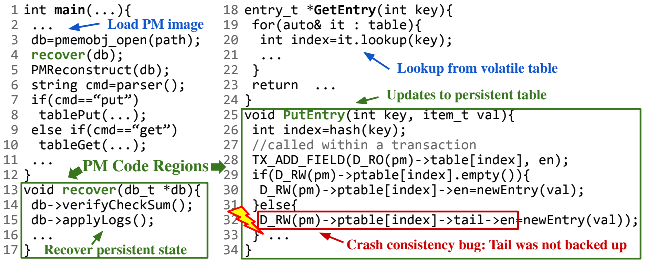


Figure 2: A general fuzzing procedure.



```
 1 int main(...){
 2  ...
 3  db=pmemobj_open(path);
 4  recover(db);
 5  PMReconstruct(db);
 6  string cmd=parser();
 7  if(cmd==“put”)
 8   tablePut(...);
 9  else if(cmd==“get”)
10   tableGet(...);
11  ...
12 }
13 void recover(db_t *db){
14  db->verifyCheckSum();
15  db->applyLogs();
16  ...
17 }
18 entry_t *GetEntry(int key){
19  for(auto& it : table){
20   int index=it.lookup(key);
21   ...
22  }
23  return  ...
24 }
25 void PutEntry(int key, item_t val){
26  int index=hash(key);
27  //called within a transaction
28  TX_ADD_FIELD(D_RO(pm)->table[index], en);
29  if(D_RW(pm)->ptable[index].empty()){
30   D_RW(pm)->ptable[index]->en=newEntry(val);
31  }else{
32   D_RW(pm)->ptable[index]->tail->en=newEntry(val));
33  } ...
34 }
```

Load PM image
PM Code Regions
Recover persistent state
Lookup from volatile table
Updates to persistent table
Crash consistency bug: Tail was not backed up

Figure 3: A buggy PM-based database (Example 2).

if conditions. Therefore, a test case generator becomes a necessity to cover such nontrivial program paths. Next, we introduce *fuzzing*, a widely-used technique for test case generation.

2.3 Requirements for Fuzzing PM Programs

A test case generator for testing PM programs should avoid introducing additional burdens on programmers, given the already complicated nature of PM programming. *Fuzzing* is a well-known technique that automatically generates test cases while minimizing programmers' effort [8, 18, 20, 24, 91]. Figure 2 shows a typical procedure of fuzzing—a fuzzer takes a set of initial test cases (or seeds), performs mutation on those test cases, executes the target program, monitors the execution statistics, and finally uses the statistics (e.g., branch coverage) to select high-value test cases. These high-value test cases will then be used in the next iteration of fuzzing. Using a fuzzer, the if-conditions in Figure 1 (Example 1) are likely to be covered. However, we identify that there are additional needs from PM programs that conventional fuzzers do not meet. Next, we provide another example of a PM crash consistency bug to motivate the new requirements.

Example 2: A buggy PM database. Figure 3 (Example 2) is a simplified example of a database based on the PMDK transaction [32]. It maintains the persistent data in PM and buffers a volatile table in DRAM for faster lookup, similar to the PM-based Redis [39]. During execution, the main() function first loads the existing persistent data that were stored on PM, which we refer to as a *PM image* (line 3), calls recover() to restore the persistence state (e.g., recover from a previous failure), and then loads the PM structures to the volatile table. Upon requests, the database calls corresponding functions, such as GetEntry() and PutEntry(). GetEntry() (line 18) looks up the key in the volatile table, and PutEntry() (line 25) updates the key-value pair in the persistent ptable. In this example, there is a crash consistency bug in PutEntry(). A new entry is appended to the tail of the indexed list in ptable when the list is not empty (line 32), whereas the previous log operation only covers the first item in the list (line 28). Thus, in case a failure happens at line 32, the update to tail can be interrupted and remains in an inconsistent state. Next, we summarize the additional requirements that traditional fuzzers need to expose PM bugs.

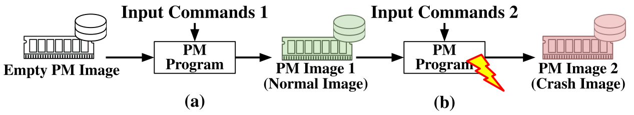


Figure 4: PM program execution procedures that generate (a) a normal image, and (b) a crash image.

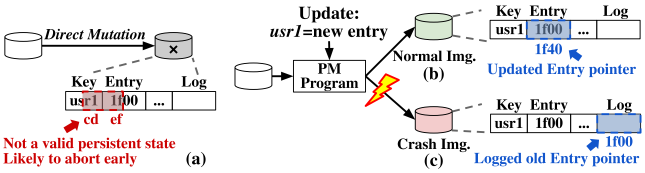


Figure 5: (a) An invalid image produced by direct mutation, (b) a normal image produced by program logic, and (c) a crash image produced by program logic.

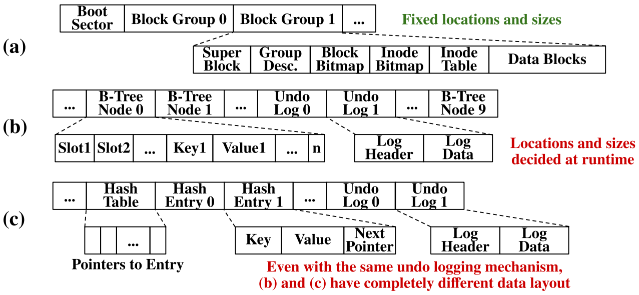


Figure 6: Persistent data layout in (a) an Ext2 file system [9], (b) a PM-based B-Tree, and (c) a PM-based database.

Requirement 1: PM images as input. A PM program typically takes PM image(s) as part of the input to maintain their persistent state, as demonstrated by the procedure in Figure 4a, and the main() function of Figure 3 (Example 2). Prior works have shown that the procedure that loads PM images can be buggy [57]. Therefore, a fuzzer for PM programs needs to generate not only the basic input commands but also PM images for testing. More importantly, the generated PM image is required to be *valid*, so that the program can execute a useful path, without failing basic image checks or triggering exceptions. However, directly fuzzing PM images through mutation is challenging—the search space of a PM image (tens of MBs) is huge, and it is hard to construct a valid PM image. Figure 5a demonstrates a PM image of a database being randomly mutated, where the mutation lies in the middle of the key and its entry pointer. Execution using this invalid image is likely to abort due to segmentation faults. Recent fuzzers have proposed to mutate file system images [44, 88] based on the preknowledge of the data layout of file systems. Figure 6a shows the simplified layout of an Ext2 file system [9], where the sizes and locations are known based on the Ext2 format. In comparison, PM programs tend to customize the way they manage persistent data. Figure 6b demonstrates the layout of Example 1, where the structures of tree nodes and logs are seemly rigid but do not follow a specific format—the nodes and undo-log entries are all allocated in the image at runtime. Figure 6c shows the layout of Example 2. Despite the use of a similar undo-logging mechanism, the data layout still differs from that of Example 1, due to their fundamental algorithmic differences.

Requirement 2: Crash images as input. PM programs are expected to be recoverable from unexpected failures. Thus, they may also load PM images caused by failures. For clarity, we refer to a PM image that is an outcome of an *uninterrupted* execution as a *normal image*, and an image that results after a *failure* as a *crash image*. Figure 4b shows a procedure, where a PM program takes an existing PM image and executes a series of input commands. During execution, a failure occurs and results in a crash image. After the program restarts after the failure, it needs to execute the recovery procedure. For example, Figure 3 (Example 2) validates the image checksum (line 14) and rolls back the prior updates using the logged data (line 15). In order to detect bugs during the recovery procedure, a crash image is also a necessity for the input test case. However, failures may happen at any point during execution, and therefore, can lead to an infinite number of crash images.

Requirement 3: Targeting PM operations. The crash consistency bugs and performance bugs are caused by PM operations, such as PM writes that modifies the state, and PM reads that loads an existing state [57]. Therefore, test case generation should be focused on program paths that perform PM operations. In real-world PM programs, such as database applications, there are both volatile and persistent code regions. In Figure 3 (Example 2), only a fraction of the code is performing PM operations, as marked by the green boxes. As such, a fuzzer should ideally focus on the interesting paths with PM operations. However, traditional coverage metrics, such as branch coverage, which are widely adopted by traditional fuzzers do not target these PM-related paths.

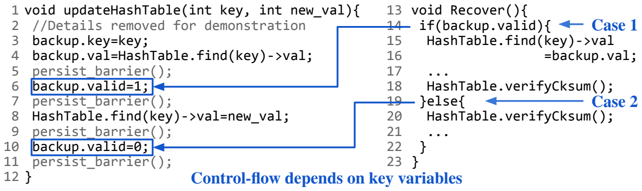
3 HIGH-LEVEL DESIGN OF PMFUZZ

So far, we have described the new requirements for fuzzing PM programs. In this work, we propose PMFuzz, a fuzzer that aims to efficiently generate test cases for debugging PM programs. Next, we discuss the challenges and our high-level design.

3.1 Normal PM Image Generation

Challenge. PM programs require that a fuzzer generates valid PM images to explore useful program paths. Conventional fuzzers are only capable of fuzzing small inputs thus do not meet this requirement. Even though file system fuzzers target large file system images, they require a well-formulated rule and image layout [44, 88]. In comparison, a PM image is not only large (e.g., tens of MBs) but also highly customized. Thus, fuzzing PM images is beyond the capability of existing fuzzers. Therefore, the *first challenge* is how can PMFuzz *efficiently generate PM images*?

Observation. As the data layout of a PM program can be largely customized, directly generating a valid PM image with permutation is hard. However, the outcome of the program logic itself always results in a *valid* persistent state. As Figure 4 demonstrates, the PM program incrementally mutates the PM image with input commands. Therefore, instead of directly fuzzing the PM image, a more effective alternative is to indirectly fuzz the input commands, which in turn will mutate the image from one valid state to another.



```
1 void updateHashTable(int key, int new_val){
2  //Details removed for demonstration
3  backup.key=key;
4  backup.val=HashTable.find(key)->val;
5  persist_barrier();
6  backup.valid=1;
7  persist_barrier();
8  HashTable.find(key)->val=new_val;
9  persist_barrier();
10 backup.valid=0;
11 persist_barrier();
12 }
```

```
13 void Recover(){
14  if(backup.valid){
15   HashTable.find(key)->val
16                 =backup.val;
17   ...
18   HashTable.verifyCksum();
19  }else{
20   HashTable.verifyCksum();
21   ...
22  }
23 }
```

Figure 7: Example of control-flow dependency between failures and the recovery procedure.

Solution. Based on this observation, our key idea is to fuzz the input commands and reuse the program logic to generate a PM image that is guaranteed to be a valid persistent state. At the high-level, the procedure of fuzzing PM images follows these steps: (1) Mutate input commands, (2) perform execution on top of an existing PM image, (3) collect the output PM image, and (4) reuse the generated PM images and repeat these steps. As PMFuzz continues to recursively operate on existing PM images, a thorough mutation on the PM image will eventually be done by the program logic itself. Figure 5b demonstrates that executing an update command creates an output PM image that has a valid mutation on the value of “Entry pointer”. Thus we conclude that leveraging program logic can efficiently generate valid PM images.

3.2 Crash Image Generation

Challenge. As PM programs are expected to recover from failures, they may also take *crash images* as the input. However, there can be an infinite number of crash images because failure can happen at any point in the program. Thus, the *second challenge* is how PMFuzz can *generate crash images* that are most effective?

Observation. Figure 7 shows an example of updating a hash table using low-level PM primitives. The program first backs up the existing `key` and `value` (line 3-4), sets the `backup` to be valid (line 6), performs the in-place update in the destination entry (line 8), and finally invalidates the `backup` (line 10). In case this procedure is interrupted by a failure, the program has a `recovery()` function. If the `backup` is valid (line 14), it rolls back the updates (line 15-16) and then verifies the checksum of the hash table (line 18). Otherwise, it verifies the checksum directly (line 20). Given a crash image that is generated during the procedure of `updateHashTable()`, the two paths during `recovery()` (as indicated by Case 1 and 2) only depend on the value of `backup.valid`. Therefore, even though a failure can happen at any point during the execution, not all resulting crash images are important for the coverage.

Solution. Inspired by the prior works that model the relationship between PM program recovery and failures [11, 57, 59, 60], we model the relationship between the program path during recovery and the prior procedure during the normal execution as a *control-flow dependency*. The significance of a crash image boils down to whether it can lead to a persistent state that affects the *control-flow* in the procedure after failure. Updates that can lead to a different control-flow are typically applied to key variables that determine the consistency state. For example, the update to `backup.valid` in Figure 7 alters the consistency state. Other examples include commit bits in undo/redo logs, and timestamps in checkpointing mechanisms. Usually, updates to such a commit variable are wrapped with *ordering points* (e.g., using a `persist_barrier()`), such that the commit variable always persists after the prior PM updates but before the successive ones.

Following this observation, our approach that reduces the number of crash images is two-fold. First, PMFuzz focuses on placing failures at ordering points to reduce the number of failure images. Second, PMFuzz also places additional failure points probabilistically, at a configurable rate. This way, even if the program is completely buggy, i.e., with a large number of misplaced ordering points, PMFuzz will still generate failure images for debugging. In both cases, crash images are generated by interrupting the execution of input commands. Therefore, all crash images maintain valid persistent states of the program. Back to the example in Figure 5, by placing a failure at the point where an undo log of the entry has been persisted but the item has not been updated, the output image will contain the old value in the "Log entry" of the crash image. During the recovery procedure, the program will use this "Log entry" to reconstruct the table.

3.3 Coverage for PM Path

Challenge. PM programs can contain various procedures but only those with PM operations can lead to crash consistency and performance bugs. The *third challenge* is how can PMFuzz efficiently generate test cases that *target PM operations*?

Observation. As prior testing works for PM programs [10, 49, 57, 58] have shown, crash consistency bugs (and also performance bugs) occur due to inappropriate PM accesses. Therefore, PMFuzz should target code regions that perform PM operations, e.g., PM reads, writes, writeback/flush primitives, and fences. However, PM reads and writes cannot be easily distinguished from regular volatile ones as they only differ in the address. Prior testing tools have been using dynamic instrumentation to keep track of these operations at the cost of tens- to hundreds-time overhead [10, 49, 57, 66]. As one of the key design principles of fuzzing is to achieve high execution efficiency, dynamic instrumentation is not a feasible choice. Despite the difficulties, we find that it is not necessary to track at the instruction granularity; instead, accesses to PM are typically wrapped with functions. As described in Section 2.1, PM libraries provide methods, such as `D_RW()` and `D_RO()`, to obtain the pointer to a PM object and to perform write/read accesses; they also provide other methods, such as `pmem_persist()`, to write-back persistent data. Therefore, the tracking granularity can be lifted to the function-level to reduce the performance overhead.

Solution. Based on the two observations, our key idea is to identify PM operations by tracking them at the granularity of PM library functions. Having PM operations being tracked, we can further design a PM-specific coverage metric to enable a *targeted fuzzing* on the PM-related program paths (see Section 4.2 for details about the mechanism). Next, we formally define the program path that contains PM operations.

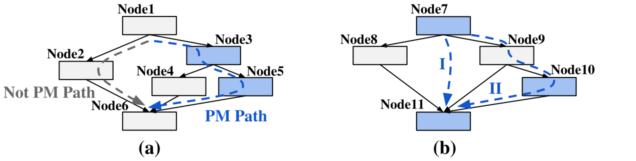


Figure 8: PM path examples (nodes in *blue* are PM nodes).

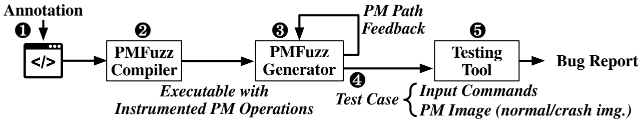


Figure 9: High-level workflow of PMFuzz.

- ***Control-flow Graph (CFG).*** A CFG of a program procedure is a directed graph represented by a tuple of $\langle N, E\rangle$; N is the set of *nodes*, where each node n represents unique program statement; $E \subset N \times N$ is the set of *edges*, where an edge e_{ij} represents execution flow between nodes n_i and n_j.
- ***Program Path (π).*** A program path in a CFG is a sequence of nodes $\pi = \langle n_0, n_1, ...\rangle$, such that there is an edge along the CFG between two consecutive nodes of the sequence.
- ***PM Node (p).*** A CFG node $p \in N$ is a PM node if it performs *at least one* PM operation.
- ***PM Path (π_{PM}).*** A PM path is a PM node sequence $\pi_{PM} = \langle p_0, p_1, ...\rangle$, such that, there is at least one edge along the CFG between two consecutive PM nodes in the sequence.

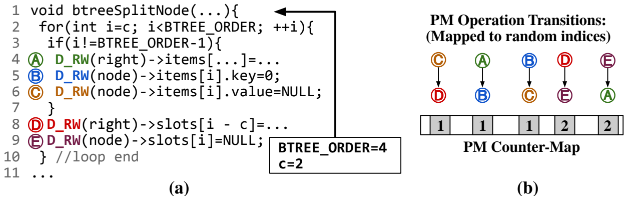
Figure 8 shows two example CFGs, where nodes in blue are PM nodes that have PM operations. Based on the definitions above, in the CFG of Figure 8a, the path of Node 1-2-6 is not a PM path due to the absence of PM operations, but the path of Node 1-3-5-6 does as it contains an edge between PM Node 3 and 5. In Figure 8b, the path of Node 7-8-11 and Node 7-9-11 are regarded as the same PM path (marked as PM Path I), because they share the same PM nodes. In comparison, the path of Node 7-9-10-11 is unique because it contains a new PM Node, Node 10 (marked as PM Path II). By tracking PM paths, PMFuzz prioritizes test cases that explore new PM paths. Therefore, PMFuzz can more efficiently generate test cases for detecting crash consistency and performance bugs.

4 IMPLEMENTATION OF PMFUZZ

In this section, we first present an overview of PMFuzz's workflow and then describe the details about the implementation.

4.1 Overview

PMFuzz is developed on top of a well-known fuzzer AFL++ [20]. It generates test cases to cover crash consistency and performance bugs in PM programs. Figure 9 shows the high-level workflow. First, PMFuzz compiler instruments the source code to keep track of PM operations (step ❶ and ❷). Then, PMFuzz takes the compiled program and performs fuzzing. The fuzzing procedure executes multiple instances of the PM program for better efficiency. During the execution of each program instance, PMFuzz monitors the coverage of the PM path and provides feedback to the fuzzing logic such that it can target PM-related operations (step ❸) that are most



```
1  void btreeSplitNode(...){
2   for(int i=c; i<BTREE_ORDER; ++i){
3    if(i!=BTREE_ORDER-1){
4 Ⓐ  D_RW(right)->items[...]=...
5 Ⓑ  D_RW(node)->items[i].key=0;
6 Ⓒ  D_RW(node)->items[i].value=NULL;
7    }
8 Ⓓ D_RW(right)->slots[i - c]=...
9 Ⓔ D_RW(node)->slots[i]=NULL;
10  } //loop end
11 ...
```

```
BTREE_ORDER=4
c=2
```

(a)

(b)

Figure 10: (a) Code instrumentation, and (b) the corresponding state of the PM counter-map for tracking PM operations.

critical to crash consistency bugs. After completing the execution of an instance, it saves the generated test case if it has explored a *new* PM path (step ❹). Each test case contains input commands and a PM image (both normal and crash images). Finally, PMFuzz sends the test cases to a testing tool (e.g., XFDetector [57] or Pmemcheck [10]) for bug detection (step ❺).

4.2 PM Operation Tracking

PMFuzz focuses on generating test cases that cover program paths that contain PM operations, such as read/write accesses, and write-back and fence primitives. As Section 3.3 has introduced, PMFuzz tracks these operations at the granularity of PM library functions. To enable this tracking, PMFuzz first performs static instrumentation using PMFuzz's compiler pass (based on LLVM [50]) and then tracks them dynamically during runtime. Next, we describe these two steps in detail.

(1) Static Instrumentation. PMFuzz tracks PM operations at function-granularity. We take an approach similar to Intel's Valgrind tool, Pmemcheck [10] and place PM operation hints inside the PMDK library. As programmers are typically agnostic about the low-level library implementation, this approach does not require any modification to programmers' application code. More specifically, PMFuzz tracks `libpmem` [34] functions that perform low-level PM operations, as well as `libpmemobj` [35] functions that provide the transaction interface. We also develop a compiler pass to support custom PM libraries. Users only need to annotate the declaration of each PM-operation function, and the compiler pass will automatically instrument the application code. Then, PMFuzz compiles the PM program and inserts a tracking function before each PM operation (i.e., library function's call site). Each tracking function is associated with a *unique ID* that marks its PM operation. Figure 10a demonstrates a simplified `btreeSplitNode()` function that highlights five PM operations, and marks their IDs with circled-letters. Next, we describe how PMFuzz keeps track of the path at runtime using the unique ID of PM operations.

(2) Dynamic Tracking. A PM path consists of a series of transitions between PM operations. Inspired by the way AFL [91] tracks branches, PMFuzz encodes the transition between two PM operations based on their unique IDs, and updates a *PM counter-map* according to the encoded value of this transition. Algorithm 1 demonstrates the transition encoding and PM counter-map update. First, the tracking mechanism reads the current PM operation's ID (`curID`), which has been assigned during compile-time (line 3). Second, it encodes the transition from the previous PM operation (with `prevID`) to the current one by XORing the two IDs (line 4).

Algorithm 1: Update to PM counter-map

```
1 begin updatePMCounterMap($Op$, $PMCounterMap$)
2   if $Op \in PMOps$ then                          // When Op is a PM operation
3     $curID = Op.ID$                               // Get ID of the current OP
4     $loc = curID \oplus prevID$                   // Encode transitions between OPs
5     $PMCounterMap[loc]++$                         // Increment counter
6     $prevID = curID \gg 1$                        // Right-shift one bit to track direction
7   return $PMCounterMap$
```

Algorithm 2: PM path prioritization

```
begin PMPathFeedback($TestCase$)
  foreach $loc \in PMCounterMap$ do
    if $unseen(PMCounterMap[loc])$ then
      $Favored = 2$                                 // High priority
    else if $diffCounter(PMCounterMap[loc])$ then
      $Favored = 1$                                 // Medium priority
    else
      $Favored = 0$                                 // Low priority
    $TestCase.Favored = Max(Favored, TestCase.Favored)$
  return $TestCase$
```

This way, a transition is encoded as an ID that serves as the index (`loc`) to a PM counter-map. The counter indicates the number of visits of this transition, as every visit of this transition increments this counter value by 1 (line 5). For lower storage overhead, each counter value is encoded with an 8-bit integer. Third, to preserve the direction of this transition, the tracking mechanism right-shifts the `curID` by 1 bit before moving toward the next PM operation (line 6). Figure 10b shows the state of a PM counter-map after `btreeSplitNode()` completes the for-loop (line 2-10), using input arguments listed in the text box. Next, we describe how PMFuzz's fuzzing logic monitors the statistics of the PM path.

4.3 Fuzzing Feedback Logic

The core fuzzing algorithm of PMFuzz provides feedback for future test case generation in order to optimize PM path coverage based on the statistics. As PMFuzz is built on top of AFL++ [20], we take a similar approach as AFL++, where we prioritize branch coverage, but also integrate an additional targeted fuzzing algorithm for PM operations. Algorithm 2 presents the prioritization algorithm of PMFuzz, which examines each location in the *PM counter-map* and sets the `Favored` value of the corresponding test case. Test cases with *unseen* PM counter-map locations are set as *high-priority*, those with significantly different counter values are set as *medium-priority*, and the remaining ones that are identical or with minor counter value differences are treated as *low-priority*. After each iteration of fuzzing, PMFuzz discards low-priority cases unless AFL++'s branch coverage logic favors them. In the next iteration of fuzzing, test cases with higher priority are more likely to be mutated to generate new test cases. This algorithm is effective but requires zero-randomness during execution, i.e., the same test case always produces the same path and PM image. Otherwise, the feedback on PM path coverage is unstable and the fuzzing outcomes are not reproducible. Next, we describe the derandomization approach.

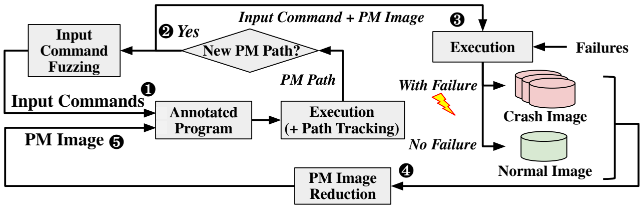


Figure 11: Fuzzing procedure of PMFuzz.

4.4 Execution Derandomization

As stated above, we notice that PM programs generally have non-deterministic execution due to three major sources of randomness. PMFuzz mitigates the randomness in the following approaches.

(1) UUID of PM Images. Each PM image created by the PMDK library [32] is associated with a *universally unique identifier* (UUID). The UUID is randomly generated during the image creation time. Therefore, it is hard to determine whether two PM images are generated from the same input or not, as the UUID in each PM image is always unique. We eliminate this randomness by overloading the UUID assignment function in PMDK (also extensible to other libraries) with our version that sets the UUID to a constant value.

(2) Address Randomization. The address randomization mechanism for both volatile and persistent addresses is another source of randomness. First, volatile addresses are randomized by the *address space layout randomization* (ASLR) technique. Because PM images may keep these random volatile pointers for convenience, we disable ASLR in the Linux kernel [7]. This method makes sure that the volatile pointers would not introduce randomness to PM images. Second, persistent addresses are randomized when the PMDK library maps a PM image to the virtual address space. We derandomize the persistent addresses by setting PMDK's environment variable `PMEM_MMAP_HINT` that forces the PM image to be mapped to the *same* virtual address every time it executes [34].

(3) External Randomness. Not only PM programs but their dependent external libraries also use time-dependent or other non-deterministic random number generators. Due to time-dependent randomness, the same input test case can lead to different execution paths. We remove this source of randomness by loading the Preeny library [76] before fuzzing. Preeny overwrites the calls to random number generators using its `derand` module, making sure that the random numbers remain the same in each run.

4.5 Detailed Fuzzing Procedure

Figure 11 demonstrates the fuzzing procedure. First, PMFuzz spawns several instances of the annotated PM program with seed test cases (step ❶). For each instance, it tracks the PM path at runtime. Upon observing a new PM path, it saves this test case for further PM image generation, and provides positive feedback to the input command fuzzing logic as described in Section 4.3 (step ❷). In the PM image generation procedure, PMFuzz generates two types of PM images: normal images and crash images (step ❸). A crash image is generated by placing failures at each ordering point and additional failures at random locations (Section 3.2); a normal image is the final outcome without any failure during the procedure. Then, the generated images go through a reduction procedure that eliminates any images that are identical to the previously generated ones (step ❹). The derandomization methods introduced in Section 4.4 ensure that the same input test case always produces the same image. PMFuzz performs image reduction by looking up the image's hash value (SHA-256) in a dictionary that keeps the hash values of all prior images. Finally, both the newly generated commands and the resulting PM images will be reused as inputs in the next iteration of fuzzing (step ❺).

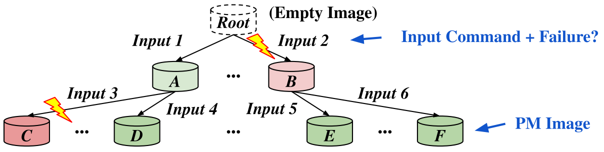


Figure 12: Tree of PM images and input commands.

4.6 Test Case Management

During fuzzing, test cases (input commands + a PM image) are generated recursively, by mutating prior test cases. PMFuzz efficiently manages the test cases by leveraging the dependencies among test cases. Figure 12 demonstrates the dependencies, where each node is a PM image (the root is an empty image), and each edge represents the input command + failure location that are used to mutate the image. The image management method serves three main purposes. First, it makes the fuzzing procedure reproducible, as each test case and its resulting PM image can be tracked by the dependency. To reproduce a particular test case, the user can simply execute the input commands on top of its parent image. Second, test case tracking allows PMFuzz to incrementally generate test cases, by loading an existing PM image and executing a set of mutated input commands (the execution time is limited to 150 ms in this design), as Section 4.5 has shown. Finally, the testing tool attached to PMFuzz (e.g., XFDetector [57] and Pmemcheck [10]) can also avoid executing redundant test cases. The testing tool only needs to execute a minimum set of test cases that cover new PM paths, without needing to start from prior test cases that contain the root image. For example, the test tool starts from test cases that contain the empty root image. Thus, to test the execution that produces image `D`, the testing tool only needs to execute `Input 4` on top of image `B`, as the execution that takes its predecessor (`Input 1` + `Root`) has been covered by the previous testing iterations.

4.7 Optimization Strategies

In this section, we introduce three major optimizations in PMFuzz that improve the fuzzing efficiency.

(1) System Call Reduction. The fuzzing procedure takes multiple system calls when opening and closing PM images. The system call overhead can be further amplified when PMFuzz executes multiple fuzzing instances simultaneously. AFL++ comes with an optimization that creates multiple fuzzing instances using its fork server's copy-on-write mechanism (via `fork()`). It would significantly reduce the system call overhead of loading PM images if we can also copy-on-write persistent data on PM images. However,

Table 1: System configuration.

CPU	Intel Xeon, 2.1GHz, 20 cores
Memory	4×16GB DDR4, 2666MT/s
	2×128GB Intel DCPMM, Interleaved, App Direct Mode
SSD	2TB, NVMe, PCI-E 3.0 ×4
OS	Ubuntu 18.04, Linux kernel v5.4
Env.	AFL++-2.63, LLVM-9, Clang-9, PMDK-1.8, Pin-3.13

this method does not apply to PM images because they are memory-mapped (i.e., a file mapped to the program's virtual address space). To take advantage of the fork server in AFL++, when the PM program is opening a PM image, we first overload the `mmap()` function with our version that copies data from PM to a location on the heap of the program. Second, we use AFL++'s fork server to create multiple fuzzing instances, while carrying the persistent data that have been loaded from the PM image to the heap. Finally, before the PM program closes the image, we overload the `munmap()` function and save the updates back to the PM image as long as the execution has discovered new PM paths (based on the method in Section 4.3). We validate this design to ensure that this optimization does not change the behavior by comparing the PM trace collected before and after applying this optimization (using Intel's Pin tool [61]).

(2) Test Case Storage. Fuzzing is a repeated process that generates a large number of test cases. Therefore, a PM device alone may not be sufficient to store all test cases. In our experiment, PMFuzz generated approximately 1.5 TB of data during a 4-hour period of fuzzing, primarily due to the PM images. Although PM images occupy a significant amount of space, we observe that the fuzzing procedure is periodical—PMFuzz takes a PM image as the input, spawns multiple fuzzer instances, saves the generated images, and starts over again by taking the newly-generated PM images as inputs. In each iteration of fuzzing, only a small fraction of PM images will be taken as inputs. And, the generated PM images will not be used until the next iteration begins. Based on this observation, PMFuzz moves the generated test cases from the PM device to a hard drive (e.g., SSD) and compresses the generated PM images (using the LZ77 [93] algorithm). PMFuzz decompresses and moves an image back to PM, only when it is selected as the input. This optimization effectively reduces the storage requirement.

5 EVALUATION

5.1 Methodology

System Configuration. We evaluate PMFuzz in a system with Intel's Cascade Lake processors and DC Persistent Memory Modules (DCPMMs), as listed in Table 1. The PM devices (i.e., DCPMMs) are configured in the App Direct Mode and mounted with the `DAX` option to bypass OS indirections.

PM Programs. To evaluate PMFuzz, we choose PM programs (listed in Table 3) built on top of Intel's PMDK (v1.8) [32] library, including simple key-value store structures [33] and real-world databases [39, 51], similar to those tested by prior works [10, 57, 58, 66]. We use PMDK's `mapcli` [36] to drive the key-value stores, and use Preeny [76] to convert the socket-based communication interface of the databases to a command-line-based version.

Table 2: Comparison points

	Input Fuzz	Img Fuzz	PM Path Opt	Sys Opt
PMFuzz (All Feat.)	Yes	Yes (Indirect)	Yes	Yes
PMFuzz w/o SysOpt	Yes	Yes (Indirect)	Yes	No
AFL++	Yes	No	No	No
AFL++ w/ SysOpt	Yes	No	No	Yes
AFL++ w/ ImgFuzz	No	Yes (Direct)	No	No

Comparison Points. PMFuzz is developed on top of AFL++ (v2.63 [1]) with the integration of state-of-the-art fuzzing techniques, including LAF-Intel [40] and AFL-Sensitive [80]. Therefore, we take AFL++ as the main baseline fuzzer. To better demonstrate the impact of each PMFuzz feature, we develop other alternative designs that are based on AFL++ and PMFuzz (listed in Table 2). The details about the features are described below.

- **Input Fuzz** (Input Fuzzing) is a feature that mutates the input commands.
- **Img Fuzz** (PM Image Fuzzing) is a feature that mutates the PM image. The PM image is *indirectly* mutated using the program itself in the comparison point of PMFuzz but is *directly* mutated in AFL++ w/ ImgFuzz. As the baseline AFL++ does not support the mutation of both the image and the command input at the same time, we only enable image fuzzing in AFL++ w/ ImgFuzz.
- **PM Path Opt** (PM Path Optimization) is a feature that enables the targeted fuzzing on PM paths (introduced in Section 4.3).
- **Sys Opt** (System-level Optimization) is a feature that reduces the system call and storage overhead (introduced in Section 4.7).

Note that, in all comparison points, we enable the derandomization techniques (described in Section 4.4) and use a list of basic commands and a PM image as the seed test case for fuzzing.

Detection Tool. PMFuzz is a test case generator that provides high-value test cases to the backend testing tools for PM programs. We leverage the most recent PM testing work XFDetector [57] as the testing tool attached to PMFuzz, which executes with PM programs and detects crash consistency and performance bugs. In addition, we use Intel's Pmemcheck [10] to detect synthetic bugs within the library (e.g., transaction, recovery, image creation, etc.).

Synthetic Bug Injection. To evaluate the effectiveness of test cases generated by PMFuzz, we place synthetic bugs in PM programs and the PDMK library, similar to the method taken by prior works [57, 58]. More specifically, we take the following approaches.

- Remove/misplace writebacks (flushes) and fences to break the persistence requirement.
- Reorder PM writes that are originally ordered with write-backs and fences, to break the ordering requirement.
- Remove/misplace backup function calls to corrupt data in transaction-based programs.
- Place semantically incorrect code to cause incorrect recovery in programs based on low-level primitives, such as setting a wrong value to the commit variables.

5.2 PM Path Coverage

Figure 13 compares the number of unique PM paths covered by PMFuzz and the comparison points during 4-hour fuzzing. We summarize the results as the following points. (1) PMFuzz achieves

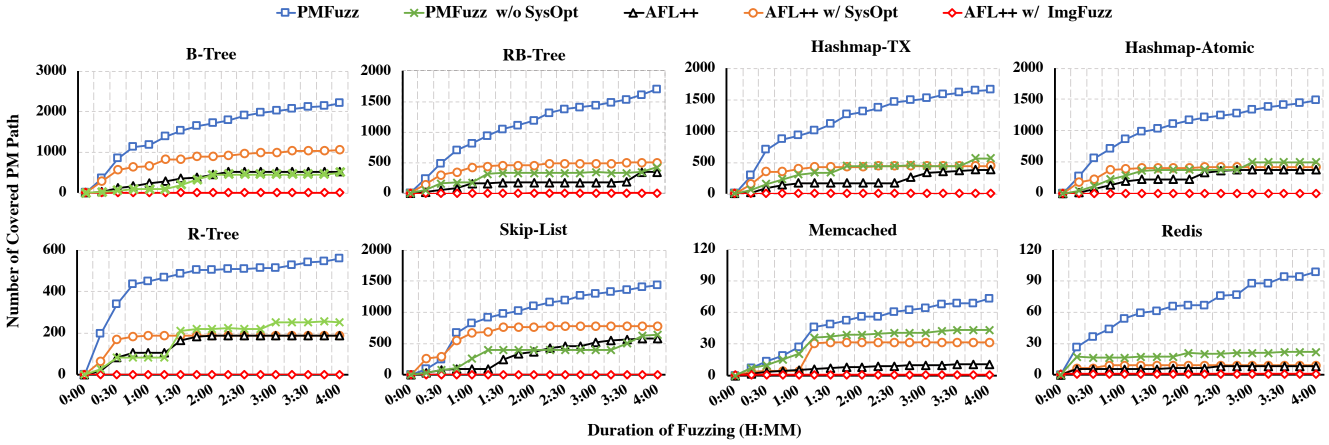


Figure 13: PM path coverage

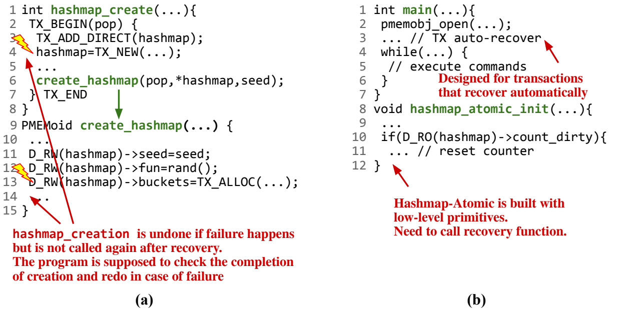
Table 3: Tested PM programs, and synthetic bug detection.

	Program Name	#Synthetic Bugs	#Covered by AFL++ SysOpt	#Covered by PMFuzz
Simple KV-store	B-Tree	17	13	17
	RB-Tree	14	10	14
	R-Tree	16	12	16
	Skip-List	12	8	12
	Hashmap-TX	21	16	21
	Hashmap-Atomic	14	10	14
DB	Memcached	17	14	17
	Redis	14	9	14

a significant increase in PM path coverage over AFL++ (Geo-mean 4.6×) because it efficiently mutates PM images, performs a targeted fuzzing on PM path, and consumes a low system overhead. (2) The PM path coverage is significantly lower without our system optimizations (PMFuzz w/o SysOpt), demonstrating that the system-level optimizations are essential to fuzzing PM programs. (3) AFL++ with system optimizations (AFL++ w/ SysOpt) outperforms AFL++ (Geo-mean 1.4×), but still cannot provide comparable coverage to PMFuzz. (4) AFL++ with PM image fuzzing (AFL++ w/ ImgFuzz) has poor coverage progress due to the large search space within PM images. Finally, the two databases, Memcached and Redis have fewer PM paths as compared to other key-value store structures. The primary reason is that only a relatively small fraction of code manages PM. Additionally, it takes much longer to execute them due to their higher complexity.

5.3 Synthetic Bug Detection

Table 3 lists the number of synthetic bugs tested and detected by PMFuzz. We compare PMFuzz with AFL++ w/ SysOpt in this experiment, as this configuration performs the best among the non-PMFuzz comparison points. We observe that PMFuzz generates test cases that detect *all* synthetic bugs, 1.4× over AFL++ w/ SysOpt, due to PMFuzz's effective PM image generation (both normal and crash images) and the focus on PM paths. Worth pointing out that the software development for PM is currently in an early stage. Therefore, the existing workloads are relatively simple. We expect that PMFuzz will show a more prominent advantage over conventional fuzzers while testing future real-world PM programs.



```
1 int hashmap_create(...){
2  TX_BEGIN(pop) {
3   TX_ADD_DIRECT(hashmap);
4   hashmap=TX_NEW(...);
5   ...
6   create_hashmap(pop,*hashmap,seed);
7  } TX_END
8 }
9 PMEMoid create_hashmap(...) {
10  ...
11  D_RW(hashmap)->seed=seed;
12  D_RW(hashmap)->fun=rand();
13  D_RW(hashmap)->buckets=TX_ALLOC(...);
14  ...
15 }
```

`hashmap_creation` is undone if failure happens but is not called again after recovery. The program is supposed to check the completion of creation and redo in case of failure

(a)

```
1 int main(...){
2  pmemobj_open(...);
3  ... // TX auto-recover
4  while(...) {
5   // execute commands
6  }
7 }
8 void hashmap_atomic_init(...){
9  ...
10  if(D_RO(hashmap)->count_dirty){
11   ... // reset counter
12 }
```

Designed for transactions that recover automatically

Hashmap-Atomic is built with low-level primitives. Need to call recovery function.

(b)

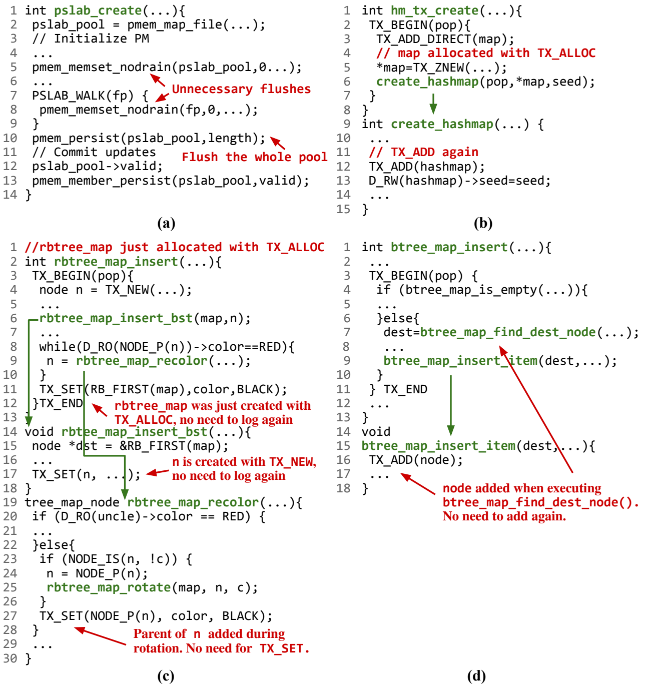
Figure 14: New crash consistency bugs found by PMFuzz: (a) Bug 1 and (b) Bug 6.

5.4 New Real-world Bugs Found by PMFuzz

Despite the fact that prior works [10, 57, 58] have intensively tested PM programs listed in Table 3, test cases generated by PMFuzz help detect new real-world bugs.

New Crash Consistency Bugs

Bug 1-5: Figure 14a is a simplified code snippet from Hashmap-TX (`hashmap_tx.c:402`), where `create_hashmap` uses a transaction (line 2-7) to allocate space and initialize the hash table. PMFuzz created two crash images before and within the allocation. When taking the crash images for the next fuzzing iteration, both of them report a segmentation fault when the program attempts to dereference the pointer to `hashmap`. We found that `hashmap_create` is called when starting with an empty PM image. In case the procedure fails, the whole creation procedure is undone by the transaction, leaving `hashmap` a `NULL` pointer. However, because the program does not call `hashmap_create` again afterward, the following execution assumes a fully initialized hash table. Other 4 transactional workloads, including B-Tree, RB-Tree, R-Tree, and Skip-List also have similar bugs during initialization. Although the prior failure-aware testing tool XFDetector [57] can detect this type of bugs with



```
 1 int pslab_create(...){
 2  pslab_pool = pmem_map_file(...);
 3  // Initialize PM
 4  ...
 5  pmem_memset_nodrain(pslab_pool,0...);
 6  ...
 7  PSLAB_WALK(fp) {
 8   pmem_memset_nodrain(fp,0,...);
 9  }
10  pmem_persist(pslab_pool,length);
11  // Commit updates
12  pslab_pool->valid;
13  pmem_member_persist(pslab_pool,valid);
14 }
```

Unnecessary flushes

Flush the whole pool

(a)

```
 1 int hm_tx_create(...){
 2  TX_BEGIN(pop){
 3   TX_ADD_DIRECT(map);
 4   // map allocated with TX_ALLOC
 5   *map=TX_ZNEW(...);
 6   create_hashmap(pop,*map,seed);
 7  }
 8 }
 9 int create_hashmap(...) {
10  ...
11  // TX_ADD again
12  TX_ADD(hashmap);
13  D_RW(hashmap)->seed=seed;
14  ...
15 }
```

(b)

```
 1 //rbtree_map just allocated with TX_ALLOC
 2 int rbtree_map_insert(...){
 3  TX_BEGIN(pop){
 4   node n = TX_NEW(...);
 5   ...
 6   rbtree_map_insert_bst(map,n);
 7   ...
 8   while(D_RO(NODE_P(n))->color==RED){
 9    n = rbtree_map_recolor(...);
10   }
11   TX_SET(RB_FIRST(map),color,BLACK);
12  }TX_END
13 }
14 void rbtree_map_insert_bst(...){
15  node *dst = &RB_FIRST(map);
16  ...
17  TX_SET(n, ...);
18 }
19 tree_map_node rbtree_map_recolor(...){
20  if (D_RO(uncle)->color == RED) {
21  ...
22  }else{
23   if (NODE_IS(n, !c)) {
24    n = NODE_P(n);
25    rbtree_map_rotate(map, n, c);
26   }
27   TX_SET(NODE_P(n), color, BLACK);
28  }
29  ...
30 }
```

rbtree_map was just created with TX_ALLOC, no need to log again

n is created with TX_NEW, no need to log again

Parent of n added during rotation. No need for TX_SET.

(c)

```
 1 int btree_map_insert(...){
 2  ...
 3  TX_BEGIN(pop) {
 4   if (btree_map_is_empty(...)){
 5   ...
 6   }else{
 7    dest=btree_map_find_dest_node(...);
 8    ...
 9    btree_map_insert_item(dest,...);
10   }
11  } TX_END
12  ...
13 }
14 void
15 btree_map_insert_item(dest,...){
16  TX_ADD(node);
17  ...
18 }
```

node added when executing btree_map_find_dest_node(). No need to add again.

(d)

Figure 15: New performance bugs found by PMFuzz: (a) Bug 7, (b) Bug 8, (c) Bug 9–11, and (d) Bug 12.

a simple test case of an empty PM image, due to the programmer's effort in understanding and annotating the source code, XFDetector did not take the buggy code region into consideration.

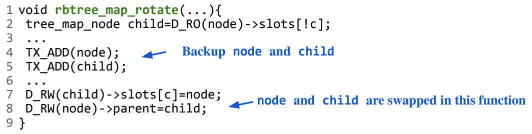
Bug 6: Figure 14c shows two functions: `main()` is a driver program for PMDK's key-value store, Mapcli (`mapcli:205`). The other function, `hashmap_atomic_init()`, is a procedure in Hashmap-Atomic (`hashmap_atomic.c:452`). This code snippet has a crash consistency bug as the `main()` function assumes all key-value store structures can automatically recover using transactions, but overlooks the low-level-primitive-based Hashmap-Atomic. Detecting this bug requires a test case that has `counter_dirty=true` (line 10), which is not easy to reach without a PM-specific test case generator.

New Performance Bugs

Bug 7: Figure 15a is a code snippet from Memcached (`pslab.c:317`) that creates a new `pslab_pool`. It starts with setting up a few metadata entries, and then flushes the whole pool. Finally, it sets a `valid` bit (surrounded with ordering points) to commit the creation (line 12). There are two redundant flushes (line 5 and 8) to the metadata as line 10 flushes the whole `pslab_pool`.

Bug 8: Figure 15b is a code snippet from Hashmap-TX that performs insertion (`hashmap_tx.c:90`). Line 12 calls a redundant `TX_ADD()` to back up a node that was previous allocated by `TX_ZNEW()` (line 5) which has logged this object.

Bug 9–11: Figure 15c is a code snippet from RB-Tree showing the procedure of an insertion function that contains three performance bugs (`rbtree_map.c:215`). **Bug 9** is at line 17 that uses `TX_SET()` to update the transaction-allocated node `n`, which introduces a redundant log operation. **Bug 10** is at line 11 that logs `RB_FIRST(map)`, which is the first entry in the tree, before performing the update. However, if the tree was just transaction-allocated (comment at line 1), it is unnecessary to log it again. **Bug 11** is at line 27 that uses `TX_SET()` to update the `parent` of node `n`, which has been backed up if `rbtree_map_recolor()` executes `rbtree_map_rotate()` first. These performance bugs can be detected by prior testing tools but require a specific test case to trigger. In particular, Bug 9 can be detected only when testing a newly allocated tree, and Bug 11 requires the if-condition at line 20 to be false but line 23 to be true.



```
1 void rbtree_map_rotate(...){
2  tree_map_node child=D_RO(node)->slots[!c];
3  ...
4  TX_ADD(node);
5  TX_ADD(child);
6  ...
7  D_RW(child)->slots[c]=node;
8  D_RW(node)->parent=child;
9 }
```

Backup node and child

node and child are swapped in this function

Figure 16: An example from RB-Tree that demonstrates the trade-off between programmability and performance.

Bug 12: B-Tree has a performance bug as shown in Figure 15d (`btree_map.c:276`). `btree_map_insert()` first finds the destination using `btree_map_find_dest_node()` and then inserts the node using `btree_map_insert_item()`. `TX_ADD()` at line 16 is unnecessary because node has been added when finding the destination (performs modification if tree-split is needed).

5.4.1 Efficiency of Test Case Generation. PMFuzz is also efficient in generating test cases that detect those bugs. To cover Bug 1–5, 7, and 8, PMFuzz only took 2 seconds of wall-clock time—as soon as the first batch of test cases was generated, since those bugs are located in the initialization step. For the rest of the bugs, Bug 9 and 10 are detected by the same case that took 91 seconds to generate; Bug 6, 11, and 12 took 37, 77, and 88 seconds, respectively. The fuzzing time was longer as covering those bugs requires relatively more complex program paths.

6 DISCUSSION

In this section, we discuss the trade-offs between programmability and performance, and then the potentials for extending PMFuzz to accommodate other types of PM software systems.

Performance Bug Trade-offs. In the PMDK library, a redundant `TX_ADD()` does *not* create additional logs. All logged locations are kept track of using a *range tree*. Before creating a new log entry, the library looks up the location in the *range tree* to make sure it has not been logged before. With this mechanism, it is safe to call `TX_ADD()` without checking conditions, such as whether the object has been backed up or allocated with a transactional interface. Nonetheless, the unnecessary range tree lookup can still lead to performance penalties (e.g., Bug 9–12). Therefore, we expect highly-optimized PM programs to avoid these redundant calls to transactional functions.

On the other hand, it is sometimes hard to completely remove performance bugs. Figure 16 shows an example from RB-Tree (`rbtree_map.c:189`), where `rbtree_map_rotate()` swaps `node` with its `child` (line 7 and 8). If this function is called multiple times, i.e., keep rotating until the tree is balanced, the two `TX_ADD()` calls (line 3 and 4) can apply to objects that have already been logged.

However, it is hard to tell whether or not a node has been logged as the rotation depends on the value of each node. Instead, it is easier to implement the rotation procedure by logging both nodes in the beginning to avoid any crash consistency issues. Therefore, we do not treat this type of issue as a performance bug.

Integration with PM Kernel Modules. There have been works that develop PM-optimized file systems for other programs to manage persistent data [17, 19, 42, 83, 86, 87]. These file systems are implemented as kernel modules but different from conventional file systems, they customize the persistent data, much like the user-space PM programs. Thus, it is hard to directly mutate their PM images. PMFuzz runs in the user-space as it is built upon AFL++ [20]. Nonetheless, it is possible to convert kernel-mode file systems into user-space programs, using libraries such as Linux Kernel Library (LKL) [69], or execute them on a virtual machine [18, 49]. This way, PMFuzz can be integrated into such frameworks to generate test cases for kernel-mode, PM-optimized file systems. We leave this direction as a future work.

Multithreading. PM programs may run in multithreaded mode for better throughput. PMFuzz is built on top of AFL++ which is thread-safe. However, multithreading introduces randomness due to various conditions of thread interleavings. As randomness prevents the fuzzer from converging to good coverage, it is not recommended to run PMFuzz with multithreading-enabled programs. On the other hand, recent works have pointed out potential persistency issues with multithreaded execution [45, 81]. PMFuzz's targeted fuzzing on PM operations can generate high-value test cases for such scenarios, with an extended focus on PM-related multithread synchronization primitives. We leave test case generation for multithreaded PM programs as a future work.

7 RELATED WORKS

In this section, we discuss the related works, including PM programming and testing, and conventional fuzzing techniques.

PM Hardware Systems. There have been a variety of hardware solutions that improve the efficiency of PM systems. For example, DPO [47], HOPS [62], and Themis [73] propose persistency models that reduce the overhead of persistence by relaxing the ordering requirements; Kiln [92], ThyNVM [72], ATOM [41], DudeTM [53], and PiCL [64] provide hardware-based mechanisms to ensure crash consistency; SCA [55], Osiris [90], Anubis [94], and Janus [56] propose secured and crash-consistent PM. Due to the new hardware primitives, these solutions may require additional programming effort to convert existing programs. PMFuzz can generate test cases to ensure correctness when adapting to a new PM platform.

PM Software Systems. PM allows for efficient access to persistent data without OS indirections. To leverage such an opportunity, there have been databases and key-value stores optimized for PM, such as PM-optimized Redis [39] and Memcached [51], Echo [4], NVMCached [84], and HiKV [85]. For better programmability, there have also been PM libraries, such as PMDK [32], Mnemosyne [79], NV-Heaps [15], and MOD [26]; and frameworks that convert legacy code to a persistent version, such as Atlas [11], NVthreads [27], iDO [54], and SFR [23]. Using the existing software interface, many applications customize their PM management [3, 12, 13, 16, 28, 67, 78, 89]. Most of these software systems require persistent data to be recoverable in case of a failure. PMFuzz can efficiently generate test cases that assist testing tools to detect crash consistency bugs.

Testing for PM Software. There have been specialized testing tools to help programmers detect crash consistency and performance bugs in PM programs. For example, Intel has developed Pmemcheck [10] and Persistence Inspector [66] on top of dynamic instrumentation tools to trace PM operations and perform testing. To improve testing efficiency and flexibility, PMTest [58] reduces the overhead of dynamic instrumentation and supports a wider range of PM software systems. XFDetector [57] further extends the testing scope by reasoning about the program execution before and after the failure. These tools make the bugs observable but still require the buggy path to be executed. Therefore, in this work, we develop PMFuzz to generate test cases that cover non-trivial program paths. We have shown that using test cases generated by PMFuzz, the existing tools (e.g., XFDetector [57] and Pmemcheck [10]) can detect more bugs. Another recent work, Agamotto [63] performs symbolic execution instead of runtime testing. In comparison, symbolic execution does not require test cases but has limitations, such as handling external libraries, dynamically-allocated memory, pointers, and loops.

Conventional Fuzzing Tools. Fuzzing is a well-known test case generation approach that requires minimal programmer's effort. Fuzzers typically prioritize conventional coverage metrics, such as branch and statement coverage. For example, a widely-used fuzzer, AFL, uses a genetic algorithm guided by branch coverage [91]. Recent fuzzers have adopted more advanced techniques, such as program transformation [40, 52, 68], Markov model [8], and machine learning [22, 70, 74, 75]. Although they are not tailored for PM programs, PMFuzz can incorporate these algorithms for better efficiency. Different from normal programs, file systems maintain persistent data on hard drives. In order to efficiently generate test cases, especially the file system image, there have been file system fuzzers that directly mutate file system images based on their data layout [44, 88]. Despite the similarities between file systems and PM programs, PM programs feature a more customized and divergent data layout, making it hard to directly generate valid PM images. Therefore, PMFuzz takes a new method that reuses the program logic to effectively generate valid, high-value PM images.

8 CONCLUSIONS

The use of persistent memory (PM) provides a substantial performance improvement but introduces additional programming complexity for the crash consistency guarantees. Prior works have provided tools to detect crash consistency and performance bugs in PM programs. However, detection of these bugs depends on test cases that execute the buggy program paths. This work provides PMFuzz, a fuzzer that efficiently generates test cases to detect non-trivial bugs in PM programs, with minimum programmer effort. Compared to the widely used fuzzer, AFL++, PMFuzz covers 4.6× more PM-related paths. Further, PMFuzz has discovered 12 new real-world bugs in PM programs that have already been intensively tested by prior works.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, Dr. Daniel Lustig, and Prof. Aasheesh Kolli for their valuable feedback. This project benefited from the stimulating discussions with Korakit Seemakhupt, Akhil Indurti, and other ShiftLab members. This work is supported by the Google fellowship program, NSF awards CCF-1845893, CCF-1822965, and CNS-2046066, and the SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP).

A ARTIFACT APPENDIX

A.1 Abstract

PMFuzz is a test case generator for PM programs, aiming to generate high-value test cases for PM testing tools. The generated test cases include both program inputs and initial PM images (normal images and crash images). The key idea of PMFuzz is to perform a targeted fuzzing on PM-related code regions and generate valid PM images by reusing the program logic. After generating the test cases, PMFuzz feeds them to the PM program and uses existing testing tools (XFDetector [57] and PMemcheck [10]) to detect crash consistency and performance bugs.

A.2 Artifact Check-list (Meta-information)

- **Program:** PMFuzz
- **Hardware:** Intel Cascade Lake and DC Persistent Memory (or emulated PM)
- **Metrics:** PM-path exploration and bug detection capability
- **Output: Test cases for PM programs**
- **Experiments:** (1) Compare the PM-path coverage of PMFuzz and AFL++ baseline. (2) Reproduce new bugs covered by PMFuzz.
- **How much disk space required (approximately):** 1 TB
- **How much time is needed to complete experiments:** The whole fuzzing procedure (including all the comparison points) will take approximately 150 hours.
- **Publicly available:** Yes
- **DOI:** 10.5281/zenodo.4322285

A.3 Description

How to access. We maintain a GitHub repository at https://pmfuzz.persistentmemory.org.

Hardware Dependencies.

- CPU: Intel Xeon Cascade Lake
- DRAM: 32 GB at least
- Persistent Memory: Intel DCPMM (or emulated PM)
- Hard Drive: 1 TB at least (to store all the compressed test cases)

Software Dependencies.

- Ubuntu 18.04 or higher
- NDCTL v64 or higher
- libunwind-dev and libini-config-dev
- Python 3.6, GNUMake >= 3.82, Bash >= 4.0, Linux Kernel version 5.4, autoconf, bash-completions, Valgrind, PMemcheck, and Anaconda

Data Sets. We tested the following workloads:

- PMDK libpmemobj examples: Btree, RTree, RBTree, Skip List, Hashmap-Atomic, and Hashmap-TX [32]
- Redis (based on PMDK libpmemobj) [39]
- Memcached (based on PMDK libpmem) [51]

A.4 Installation

This artifact has the following structure:

- `include/`: Runtime for pmfuzz (`libpmfuzz.so` and tracing functions for XFDetector).
- `inputs/`: Inputs used as seeds for the PMFuzz.
- `scripts/`: Installation and artifact-evaluation scripts.
- `src/pmfuzz`: Source for our test case generator.
- `vendor/{pmdk,memcached,redis}`: Workloads.
- `vendor/{pmdk,memcached,redis}-buggy`: Workloads with annotations for bug reproduction.
- `vendor/xfdetector`: Source for XFDetector testing tool.
- `preeny`: git submodule for Preeny tools [76].

Setup Environment. PMFuzz requires the environment variable for `PIN_ROOT` and `PMEM_MMAP_HINT` are set before execution. To set these variables, please execute the following commands:

```
export PIN_ROOT=<PMFuzz Root>/vendor/pin-3.13
export PMEM_MMAP_HINT=0x10000000000
```

A PM device (in App Direct mode) also needs to be mounted at `/mnt/pmem0` with the `DAX` option enabled. To do so, please execute the following command:

```
sudo mount -o dax /dev/pmem0 /mnt/pmem0
```

It also requires disabling ASLR and core dump notifications. To disable them, please execute the following commands (need to execute again after power cycle):

```
echo core | sudo tee /proc/sys/kernel/core_pattern
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Setup Software Dependencies. To run PMFuzz, please make sure that all the dependencies are installed (Section A.3). If some dependencies are not met, our script can install them:

```
cd <PMFuzz Root>
./scripts/install-dependencies.sh
```

NOTE: This command will remove the existing `libndctl` and update it to the required version.

Setup Python Environment. In addition to the basic dependencies, PMFuzz requires a Python 3.6 environment, together with several Python packages. To install them, please execute the following commands:

```
pip3 install -r src/pmfuzz/requirements.txt
```

Install PMFuzz and PM Workloads. To download the correct version of LLVM, compile PMFuzz's runtime, AFL++, and all the workloads, please execute the following commands (follow the order in the listing):

```
make # Compiles our tool and PDMK
make redis memcached # Compiles other workloads
```

A.5 Experiment Workflow

The core functionality of PMFuzz is the fuzzing logic that generates test cases for PM programs. To Run the workloads using PMFuzz, please use the `run-workloads.sh` script which invokes PMFuzz with the correct arguments to run a workload. The script takes input in the following format:

```
scripts/run-workloads.sh \
    <workload name> <config name> <output dir>
```

These commands will run PMFuzz with the configuration used for the evaluation section. The script by default uses 38 CPU cores. To adjust that, please modify line 69-72 of the script. Note that the design point that generates PM images through fuzzing is supported with a separate script:

```
scripts/run-imgfuzz.sh <workload> <output dir>
```

For example, to run PMDK's btree workload in the baseline configuration, run the following command:

```
scripts/run-workloads.sh btree baseline /tmp/
```

Running this command will create the directory
`/tmp/btree,baseline` with all generated test cases and images.

A.6 Evaluation and Expected Result

The main evaluation includes the performance evaluation (Section A.6) that compares the PM path coverage (defined in Section 3.3), and reproduction of the new real-world bugs found using our generated test cases (Section A.6).

Performance Evaluation. Considering the execution time, it is recommended to run PMFuzz using more than one machine, each of which runs a fraction of workloads and design points. Before running any command, please make sure that the python environment is correctly set up, all the dependencies are installed, and the current working directory (CWD) is the root of the PMFuzz artifact repository. All PMFuzz scripts also read the environment variable `JOBS` to run make in parallel (with the default value of `-j8`). To set this variable, export it in the shell session:

```
export JOBS=-j$(nproc)
```

To make sure that the script can communicate with the hosts, please edit the variables `user`, `hosts`, `dests`, and `ssh_cmds` according to your environment in both `scripts/run-artifact-perf.py` and `scripts/show-artifact-perf-results.py`.

To run performance evaluation and automatically schedule fuzzing jobs across all the servers, please run the following commands on one of the machines:

```
./scripts/run-artifact-perf.py
```

The script will now `ssh` to other servers and start fuzzing processes. When all the fuzzers have completed, the script will exit with a message of "All Done". To plot the results (reproduce Figure 13), please execute the following commands:

```
scripts/show-artifact-perf-results.py
python -m http.server 1010
```

After completing these steps, the result will be plotted as
`evaluation-perf-result.png`

Reproducing New Real-world Bugs. To detect real bugs that we reported, please run the following script:

```
./scripts/test-real-bugs.sh [1..12]
```

where `[1..12]` corresponds to the bug IDs in Section 5.4. For example, to detect Bug 1 in Hashmap-TX, please execute the following command:

```
./scripts/test-real-bugs.sh 1
```

A.7 Experiment Customization

Execute PMFuzz without Script. To run PMFuzz directly, without using any driver scripts, please run the following command:

```
./src/pmfuzz/pmfuzz-fuzz.py \
    <Input dir> <Output dir> <Config file>
```

- `<Input dir>`: PMFuzz uses test cases from this directory as the fuzzer's seed input.
- `<Output dir>`: All the generated outputs will be placed in this directory.
- `<Config file>`: A configuration file that specifies the fuzzing target and different PMFuzz parameters.

PMFuzz Configuration. PMFuzz uses a YML-based configuration to set different parameters for fuzzing (including the fuzzing target). To write a custom configuration, please follow one of the existing examples in the `src/pmfuzz/configs/examples/` directory.

REFERENCES

[1] AFLplusplus. American fuzzy lop plus plus (afl++). https://github.com/AFLplusplus/AFLplusplus/tree/2.63c.

[2] ARM. ARM architecture reference manual ARMv8, for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf, 2018.

[3] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, 2018.

[4] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)*, 2013.

[5] Jeff Barr. Now available – Amazon EC2 high memory instances with 6, 9, and 12 TB of memory, perfect for SAP HANA. https://aws.amazon.com/blogs/aws/now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfect-for-sap-hana/, 2018.

[6] Nan Boden. Available first on Google cloud: Intel Optane DC persistent memory. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory, 2018.

[7] Michael Boelen. Linux and ASLR: kernel/randomize_va_space. https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/, 2016.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[9] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[10] Eduardo Carellan. Discover persistent memory programming errors with pmemcheck. https://software.intel.com/content/www/us/en/develop/articles/discover-persistent-memory-programming-errors-with-pmemcheck.html, 2018.

[11] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014.

[12] Shimin Chen and Qin Jin. Persistent B+-Trees in non-volatile main memory. In *The Proceedings of the VLDB Endowment*, 2015.

[13] P. Chi, W. Lee, and Y. Xie. Adapting B+-Tree for emerging nonvolatile memory-based main memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

[14] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[15] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[16] Nachshon Cohen, David T. Aksun, and James R. Larus. Object-oriented recovery for non-volatile memory. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

[17] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[18] David Drysdale. Coverage-guided kernel fuzzing with syzkaller.

[19] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *European Conference on Computer Systems (EuroSys)*, 2014.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[21] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous checkpointing of HTM transactions in NVM. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2017.

[22] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[23] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[24] Google. OSS-Fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz.

[25] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *USENIX Annual Technical Conference (ATC)*, 2019.

[26] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally ordered durable datastructures for persistent memory. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[27] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[28] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *USENIX Annual Technical Conference (ATC)*, 2017.

[29] Intel. https://software.intel.com/content/www/us/en/develop/articles/code-sample-enable-your-application-for-persistent-memory-with-mysql-storage-engine.html.

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

[31] Intel. Key/value datastore for persistent memory. https://github.com/pmem/pmemkv.

[32] Intel. Persistent memory programming. https://pmem.io/.

[33] intel. Pmdk examples. https://github.com/pmem/pmdk/tree/stable-1.8/src/examples/libpmemobj.

[34] Intel. PMDK man page: libpmem. https://pmem.io/pmdk/manpages/linux/master/libpmem/libpmem.7.html.

[35] Intel. PMDK man page: libpmem. https://pmem.io/pmdk/manpages/linux/master/libpmemobj/libpmemobj.7.html.

[36] Intel. Pmdk mapcli. https://github.com/pmem/pmdk/blob/master/src/examples/libpmemobj/map/mapcli.c.

[37] Intel. Btree: remove not needed snapshot (PMDK). https://github.com/pmem/pmdk/commit/b9232407a794040102e769ed98b967d797c173fd/#diff-c1ecccb1fea662a18db843553f5a09b00494692dc699f11c784b65d9a22535f8, 2018.

[38] Intel. Intel 64 and IA-32 architectures software developer's manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2019.

[39] Intel. Redis. https://github.com/pmem/redis/tree/3.2-nvml, 2019.

[40] Laf Intel. Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016.

[41] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *Proceedings of The 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[42] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[43] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using NVM as virtual memory. In *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.

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

[45] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[46] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[47] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[48] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[49] Philip Lantz, Dulloor Subramanya Rao, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *USENIX Annual Technical Conference (ATC)*, 2014.

[50] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[51] Lenovo. Memcached-pmem. https://github.com/lenovo/memcached-pmem, 2018.

[52] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*. Acm, 2017.

[53] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[54] Qingrui Liu, Joseph Lzraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[55] S. Liu, A. Kolli, J. Ren, and S. Khan. Crash consistency in encrypted non-volatile main memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[56] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. Janus: Optimizing memory and storage support for non-volatile memory systems. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[57] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[58] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[59] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent Computing: Challenges and Opportunities. In *2nd Summit on Advances in Programming Languages (SNAPL)*, 2017.

[60] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[61] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[62] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[63] Ian Neal, B. Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, S. Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[64] Tri Nguyen and David Wentzlaff. PiCL: A software-transparent, persistent cache log for nonvolatile main memory. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[65] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[66] Kevin Oleary. How to detect persistent memory programming errors using Intel Inspector - Persistence Inspector. https://software.intel.com/content/www/us/en/develop/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.html, 2018.

[67] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[68] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. *In Proceedings of the IEEE Symposium on Security & Privacy*, 2018.

[69] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The linux kernel library. In *9th RoEduNet IEEE International Conference*, 2010.

[70] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, pages 1–10, 2017.

[71] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. Programming for non-volatile main memory is hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys)*, 2017.

[72] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[73] S. M. Shahri, S. Armin Vakil Ghahani, and A. Kolli. (Almost) Fence-less persist ordering. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[74] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. MTFuzz: fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[75] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. *In Proceedings of the IEEE Symposium on Security & Privacy (SP)*, 2019.

[76] Yan Shoshitaishvili. Preeny. https://github.com/zardus/preeny/.

[77] Usharani Upadhyayula. Quick start guide: Provision intel optane dc persistent memory. https://software.intel.com/content/www/us/en/develop/articles/quick-start-guide-configure-intel-optane-dc-persistent-memory-on-linux.html, 2019.

[78] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST)*, 2011.

[79] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memeory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[80] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[81] William Wang and Stephan Diestelhorst. Persistent atomics for implementing durable lock-free data structures for non-volatile memory (brief announcement). In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

[82] Song Wu, Fang Zhou, Xiang Gao, Hai Jin, and Jinglei Ren. Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications. *ACM Trans. Archit. Code Optim.*, 15(4), January 2019.

[83] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[84] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. NVMcached: An NVM-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (ApSys)*, 2016.

[85] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *USENIX Annual Technical Conference (ATC)*, 2017.

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

[87] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[88] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

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

[90] M. Ye, C. Hughes, and A. Awad. Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

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

[92] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[93] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 1977.

[94] K. A. Zubair and A. Awad. Anubis: Ultra-low overhead and recovery time for secure non-volatile memories. In *ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.