PMFuzz: Test Case Generation for Persistent Memory Programs

Sihang Liu[†][*], Suyash Mahar[¶][*], Baishakhi Ray[‡], and Samira Khan[†]
[†]University of Virginia [¶] University of California, San Diego [‡]Columbia University

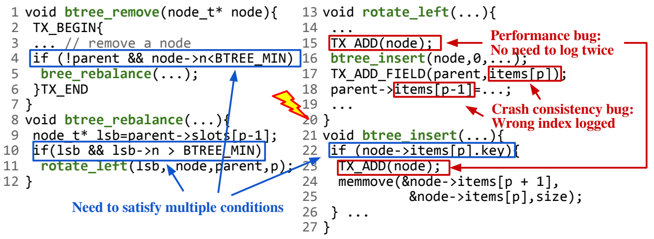
1. Background and Motivation

Persistent memory (PM) technologies, such as Intel's Optane [12], provide a class of high-performance and byte-addressable memory. PM programs directly access persistent data through the memory bus, without using software intermediaries, blurring the difference between memory and storage. Programs such as databases [15, 17, 19] and key-value stores [3, 13, 27, 28], as well as those that customize persistent storage [2, 6–8, 11, 23, 26, 29] can significantly benefit from PM. These programs generally require that the persistent data in PM can recover to a consistent state in the event of a failure (e.g., 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 persistent *prior* to the updated tail pointer that points to the new node. To prescribe the order in which writes become persistent, PM hardware platforms have introduced writeback and fence instructions, such as `CLWB` and `SFENCE` in x86 systems [16]. Building on top of these low-level primitives, there have been works that provide PM libraries, such as failure-atomic transactions from Intel's PMDK [14], that improves the programmability. Nonetheless, programmers still need to understand the crash consistency guarantees from the library and the desired failure-recovery mechanism in their programs. Prior works pointed out that programming for PM systems is error-prone [5, 18, 20, 21, 24]. 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 guarantee, which is referred to as a *crash consistency bug*. Whereas, *overuse* of these functions, such as placing redundant `SFENCE`'s or creating unnecessary backups, can degrade the performance, which is referred to as a *performance bug*.

Existing tools detect these PM bugs [5, 18, 20–22] by tracing low-level PM operations and checking whether the traces violate the persistence and ordering guarantees to ensure crash consistency. However, these tools detect a bug *only when* the buggy procedure is executed. They will miss the bug if the buggy path is *not executed* during testing. Figure 1 shows a B-Tree derived from one of the PMDK examples [14]. The code snippet shows a function that removes a node in the tree and then rebalances the tree after removal. To ensure crash consistency, it wraps the transaction procedure with a pair of `TX_BEGIN` and `TX_END`, and logs PM objects through `TX_ADD()/TX_ADD_FILED()`. After removing a tree node, the program tries to rebalance the tree by calling `btree_rebalance()` that performs `btree_rotation()` and `btree_insert()`. We place to synthetic bugs to this example as indicated by the red arrows: A crash consistency bug at line 17 where the transaction logs a wrong item, and a performance bug at line 23 where the transaction logs the same item twice. Even though existing testing tools are capable of detecting these types of bugs, they require a test case that triggers a specific series of if-conditions during the execution, as indicated by the blue arrows. Thus, to effectively expose the PM bugs, we need a test case generator that generates test cases and explore the buggy program paths.

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



```
1  void btree_remove(node_t* node){
2   TX_BEGIN{
3   ... // remove a node
4   if (!parent && node->n<BTREE_MIN)
5    bree_rebalance(...);
6   }TX_END
7  }
8  void btree_rebalance(...){
9   node_t* lsb=parent->slots[p-1];
10  if(lsb && lsb->n > BTREE_MIN)
11   rotate_left(lsb,node,parent,p);
12 }
13 void rotate_left(...){
14  ...
15  TX_ADD(node);
16  btree_insert(node,0,...);
17  TX_ADD_FIELD(parent,items[p]);
18  parent->items[p-1]=...;
19  ...
20 }
21 void btree_insert(...){
22  if (node->items[p].key){
23   TX_ADD(node);
24   memmove(&node->items[p + 1],
25            &node->items[p],size);
26  } ...
27 }
```

Figure 1: A buggy PM-based B-Tree

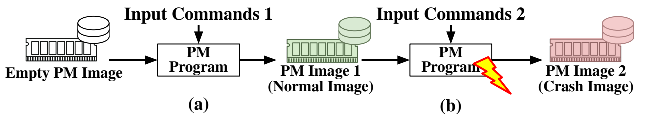


Figure 2: PM program execution procedures that generate (a) a normal image, and (b) a crash image.

2. Test Case Generation for PM Programs

Due to the already complicated programming for PM systems, a test case generator ideally should not introduce additional programmer's effort. *Fuzzing*, a test case generation method, perfectly meets this need—it is effective but only requires minimum knowledge about the program [1, 4, 9, 10, 30]. At a high-level, a fuzzer *iteratively* mutates existing test cases to generate new ones and keeps executing the program with the new test cases. We identify that additional requirements need to be met to efficiently generate test cases for PM programs.

Requirement 1: PM Image as Input. First, PM programs maintain the persistent data using PM devices, where the persistent state is first manipulated by the program and remains after program termination. Typically, this is achieved by maintaining a *PM image* in a DAX file system [25]. Therefore, a PM program takes not only executes *regular program inputs* (e.g., a command that inserts a key-value pair) but also takes an existing persistence state maintained by *PM image(s)* , as described in Figure 2a. As the procedure of loading an existing PM image may also lead to crash consistency bugs [18, 20], it is necessary for a fuzzer to generate test cases containing PM images. Conventional fuzzers are able to generate the regular input commands but are ineffective in generating PM images. Performing mutation within a large search space of an image (at least tens of MBs) is inefficient. And, as PM programs typically customize data layout in the PM image, a direct mutation is also likely to generate invalid PM images. For example, a PM image with illegal pointers can cause program to abort during initialization without exploring any useful path.

Requirement 2: Crash Image as Input. A PM image can also be a result of a procedure interrupted by a failure, as demonstrated in Figure 2b. We refer to PM images generated by a normal, uninterrupted procedure as *normal images*, and those caused by crashes as *crash images*. As required by the crash consistency guarantee, PM programs are also expected to recover from such crash images. Therefore, besides the normal images, the fuzzer also needs to generate *crash images* for a thorough testing. However, the difficulty is that failures can occur at any point during execution, leading to an infinite number of crash images.

Requirement 3: Targeting PM Operations. PM programs may contain procedures for different purposes, not limited to PM-related procedures, especially in large real-world applications. On the other hand, only PM operations are critical to crash consistency bugs, such as PM writes that modify the state, and PM reads that loads an existing state [20]. However, conventional coverage metrics adopted by existing fuzzers are not specific to PM operations. For example, the widely used statement coverage metric treats each code region evenly as it optimizes for covering more lines of code. Thus, the existing fuzzers are not efficient in generating test cases that cover PM operations which are most critical to crash consistency bugs. Therefore, an efficient fuzzer for PM programs is required to target PM operations.

3. Key Insights

Given the above requirements, existing fuzzers are insufficient. Here, we introduce PMFuzz, a new fuzzing tool that generates test cases to detect crash consistency and performance bugs in PM programs. Next, we describe our high-level ideas.

Efficient PM Image Generation. An effective fuzzer should generate valid PM images. Despite the difficulties in direct mutation, we observe that different PM images are essentially outcomes of the program execution with different input commands. Therefore, our key idea is to use the program itself to *mutate* an existing PM image. PMFuzz incrementally generates the PM image by fuzzing the input commands that modify the images. As the fuzzing procedure continues, the PM image will eventually be holistically mutated.

Prioritizing Important Crash Images. Even though failure can occur at any point during execution, we observe that the recovery procedure typically depends on a few key data blocks in the crash image. In an undo-logging mechanism, the program performs the following steps: back up the old data, set the valid bit of the undo log, perform in-place update, and finally unset the valid bit. In case of a failure, the recovery code checks the valid bit to determine whether the undo log or the in-place update has the consistent data. Therefore, to cover both the path that applies the undo log and keeps the in-place update, only two crash images are necessary—one with valid being `true` and another `false`. Therefore, our key idea is to minimize the number of crash images by only generating those that affect the control-flow during failure-recovery.

Targeted Fuzzing for PM Path. PM operations that incorrectly manage persistent data lead to PM bugs that we concern. Thus, to achieve high coverage of these bugs, the fuzzer needs to perform a *targeted fuzzing* on program paths that access PM. To enable this prioritization, we first define the *PM path* as a path consisting of program statements with PM operations. During fuzzing, PMFuzz monitors the statistics of PM path, and prioritizes test cases that cover *new* PM paths. By focusing on PM path, PMFuzz can efficiently generate test cases that target crash consistency and performance bugs.

4. Main Artifact

This work provides PMFuzz, a fuzzer that automatically generates high-value test cases to detect crash consistency bugs in PM programs. The source of PMFuzz is *publicly available* at: `https://github.com/Systems-ShiftLab/PMFuzz`. We implement PMFuzz on top of a well-known fuzzer, AFL++ [1] by incorporating our key insights, and evaluate PMFuzz in a real PM system with common PM programs, including key-value stores [14] and databases [17, 19].

5. Key Results and Contributions

- PMFuzz is the first fuzzing work that aims to generate test cases for PM programs.
- PMFuzz is tailored for PM programs by efficiently generating both normal PM images and crash images, and performing targeted fuzzing on PM program paths.
- Our evaluation shows that PMFuzz covers an average of 4.6× more PM paths over AFL++, within a 4-hour fuzzing.
- The generated test cases detect 8 new bugs in PM programs that have already been extensively tested by prior works.

6. Why ASPLOS

PMFuzz generates test cases to test the crash consistency guarantee of programs designed for persistent memory systems. It lies in the areas of systems, architecture, and testing.

7. Citation for Most Influential Paper Award

This paper advanced the testing of persistent memory programs. It designed and implemented a test case generation framework to detect crash consistency bugs, by efficiently generating inputs tailored for persistent memory programs.

References

[1] AFLplusplus. American fuzzy lop plus plus (AFL++). https://aflplus.plus/.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] Nachshon Cohen, David T. Aksun, and James R. Larus. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2018.

[9] David Drysdale. Coverage-guided kernel fuzzing with syzkaller. https://lwn.net/Articles/677764/, 2016.

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

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

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

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

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

[15] Intel. Code sample: Enable your application for persistent memory with MySQL storage engine. https://software.intel.com/content/www/us/en/develop/articles/code-sample-enable-your-application-for-persistent-memory-with-mysql-storage-engine.html, 2019.

[16] 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.

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

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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