Cross-Failure Bug Detection in Persistent Memory Programs[*]

Sihang Liu[†], Korakit Seemakhupt[†], Yizhou Wei[†], Thomas Wenisch[★], Aasheesh Kolli[‡§], and Samira Khan[†]
[†]University of Virginia [★]University of Michigan [‡]Penn State University [§]VMware Research

1 Introduction

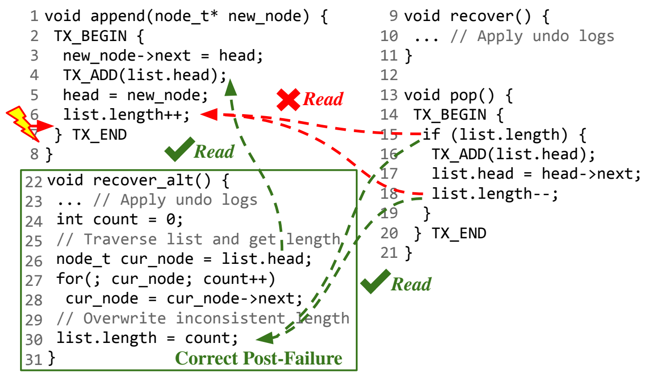
Persistent memory (PM) technologies, such as Intel's Optane memory, deliver high performance, byte-addressability, and persistence, allowing programs to directly manipulate persistent data in memory without any OS intermediaries. To leverage this new type of memory, software systems are being developed, such as data bases and key-value stores [4, 5], PM-optimized file systems [1], and PM libraries [3]. An important requirement of these programs is that persistent data must remain consistent across a failure, which we refer to as the crash consistency guarantee. Due to volatile caches and reordering of writes within the memory hierarchy, programs need to carefully manage the order in which a write becomes persistent. For example, commonly used undo logging mechanisms ensure that logs of data values must be written to PM *before* a corresponding update happens. The hardware platform has provided new primitives to enforce the order of PM writes (e.g., `CLWB` and `SFENCE` from x86). For better programmability, libraries for PM wrap these low-level primitives and provide a simpler, transactional interface. However, maintaining crash consistency is not trivial, as programmers need to have a good understanding of the hardware primitives and/or the library semantics.

1.1 Background on PM Programming

Prior works have provided testing tools [2, 7, 8], covering bugs due to misuse of either the hardware primitives or libraries during normal (fault-free) execution, e.g., when the program is mutating persistent objects. As required by the crash consistency guarantee, a correct PM program is expected to recover to a consistent state and resume execution after failure. Thus, a testing tool also needs to cover the *entire* procedure, spanning both normal execution and recovery/resumption. We divide this execution procedure into two stages with respect to the failure: the *pre-failure* and the *post-failure* stages. These two stages must work collaboratively to guarantee crash consistency. Any incorrect interaction can lead to an inconsistent recovery. Next, we use two examples to illustrate incorrect cross-failure interactions.

Example 1: Inconsistency in the post-failure execution. Figure 1 shows a snippet of code that appends a `new_node` to a persistent linked list, which uses a transaction (indicated by `TX_BEGIN` and `TX_END`) to ensure crash consistency. Within the transaction, it adds the current PM object to an undo log with a `TX_ADD()` function (line 4) before update, such that there is a consistent copy for rollback in case of failure. However, the program does not add `length` to the undo log. As a result, if a failure happens between line 6 and 7, it is unknown if the `length` of the linked list has persisted. Whether or not this inconsistent `length` can lead to a bug depends on the post-failure execution.

A *naive* but buggy implementation of the `recover()` function first rolls back the incomplete transaction with undo logs. Without having `length` recovered, the `pop()` function in the resumption execution reads an inconsistent value (as indicated by the red arrows). A *correct* implementation, `recover_alt()`, obtains the number of nodes by traversing the linked list and *overwriting* the `length` with the correct value (line 27-30). During traversal, the program reads from the consistent value of `head` as it has been backed up by the transaction (indicated by the green arrow). Thus, `length` ends up being consistent after recovery.

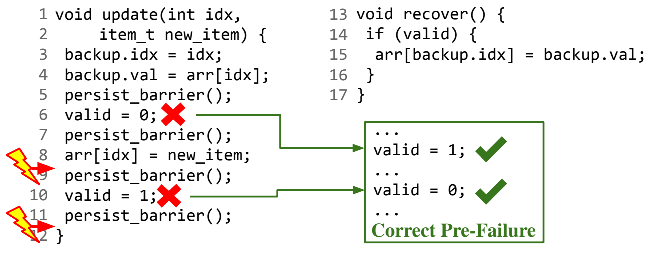


```
 1 void append(node_t* new_node) {
 2  TX_BEGIN {
 3   new_node->next = head;
 4   TX_ADD(list.head);
 5   head = new_node;
 6   list.length++;
 7  } TX_END
 8 }

 9 void recover() {
10  ... // Apply undo logs
11 }
12
13 void pop() {
14  TX_BEGIN {
15   if (list.length) {
16    TX_ADD(list.head);
17    list.head = head->next;
18    list.length--;
19   }
20  } TX_END
21 }

22 void recover_alt() {
23  ... // Apply undo logs
24  int count = 0;
25  // Traverse list and get length
26  node_t cur_node = list.head;
27  for(; cur_node; count++)
28   cur_node = cur_node->next;
29  // Overwrite inconsistent length
30  list.length = count;
31 }
```

Figure 1. Inconsistency during post-failure execution (Example 1).



```
 1 void update(int idx,
 2      item_t new_item) {
 3  backup.idx = idx;
 4  backup.val = arr[idx];
 5  persist_barrier();
 6  valid = 0;
 7  persist_barrier();
 8  arr[idx] = new_item;
 9  persist_barrier();
10  valid = 1;
11  persist_barrier();
12 }

13 void recover() {
14  if (valid) {
15   arr[backup.idx] = backup.val;
16  }
17 }
```

```
...
valid = 1;
...
valid = 0;
...
```

Figure 2. Inconsistency during pre-failure execution (Example 2).

Example 2: Inconsistency in the pre-failure execution. Figure 2 shows a snippet of code that updates a location `idx` in a persistent array (`arr`). For recovery, the `update()` function first backs up the old data and the updated index (line 3-4), issues a `persist_barrier()`, a sequence of "`CLWB;SFENCE`", to persist the backup, and sets a `valid` bit (line 6). Then, it performs the in-place update to the array (line 8). Finally, it persists the updates and resets the `valid` bit (line 9-11). Even with correctly placed `persist_barrier()`'s, the pre-failure code is *semantically incorrect* as `valid` is set to wrong values (corrections in the green box). After failure, the `recovery()` function either leaves the potentially *non-persisted* update (failure happens at line 8), or restores a stale backup (failure happens at line 12). Therefore, the recovery is *always* incorrect in either case.

1.2 Need for An End-to-end Cross-failure Detection

Prior works [2, 7, 8] cannot precisely detect these bugs. In Example 1, though `recover_alt()` is correct, existing tools can raise a false alarm even; in Example 2, the bug cannot be detected by existing tools as `persist_barrier()`'s are correctly placed. The fundamental reason is that they lack an *end-to-end* test that relates both stages. Hence, detecting crash consistency bugs should cover both the pre- and post-failure stages to ensure they work seamlessly for a consistent recovery. Next, we categorize cross-failure bugs.

[*]This work was originally published at ASPLOS'20 [6]. Link to paper: https://dl.acm.org/doi/10.1145/3373376.3378452

2 Categorization of Cross-failure Bugs

We identify two categories of cross-failure bugs: cross-failure race (and its benign scenario) and cross-failure semantic bug.

Type I: Cross-failure Race: *The post-failure execution reads from data modified by the pre-failure execution that is not guaranteed to be persisted before the failure.* The first type of cross-failure race covers the most general case of inconsistent data on PM—it happens when writes to PM are not guaranteed to be written back before a failure. As data may not be persistent, the post-failure execution can read partially updated data, leading to inconsistencies after failure. Reading the variable `length` during recovery in Figure 1 is an example of a cross-failure race as `length` is not guaranteed to be persisted before failure. Its unknown persistence status can lead to uncertainties during the post-failure stage.

Type I (Exception): Benign Cross-Failure Race: *A program intentionally reads from potentially non-persisted data modified by the pre-failure execution, without causing inconsistencies.* Cross-failure races can cause inconsistencies, however, *not all* cross-failure races lead to inconsistencies. Instead, it is sometimes necessary to read potentially stale data to correctly recover from a failure, analogous to the inherent data races on synchronization primitives. We refer to such intentional reads to inconsistent data as a *benign cross-failure race.* For example, reading the `valid` bit in Figure 2 (the corrected version) during the post-failure stage is a benign cross-failure race, as `valid` can either be 0 or 1. Thus, reading its value has a deterministic outcome, and enables the recovery procedure to determine which version (backup or original location) is consistent.

Type II: Cross-failure Semantic Bug: *The post-failure execution reads from data updated during the pre-failure stage that is semantically inconsistent according to the program.* This type of cross-failure bug covers inconsistencies defined by the program semantics. Even if a PM location has been persisted before failure, it can still be semantically inconsistent if it violates the corresponding data consistency requirements. For example, crash consistency mechanisms typically keep two versions of data: a consistent version for recovery and another for the current update. The version that is regarded as consistent by the crash consistency mechanism can be safely read during the post-failure execution. In contrast, the inconsistent version should be discarded or overwritten.

Goal. In this work, we seek to test crash consistency guarantees holistically, considering both pre- and post-failure execution stages. Next, we describe the high-level design of XFDetector.

3 High-level Design of XFDetector

Figure 3 presents XFDetector's workflow, including (step ❶) program annotation, (step ❷) failure injection during compile time, and (step ❸) failure triggering, (step ❹-❺) pre- and post-failure tracing, and (step ❻) detection during runtime. The detection mechanism consists of two main components: one that determines the consistency of PM data, and one that injects failures into program execution to trigger pre- and post-failure stages.

Data Consistency. Detecting cross-failure bugs requires determining whether the data read by the post-failure stage is consistent. However, data consistency changes as a program manipulates persistent data. Therefore, to capture updates, XFDetector traces PM operations (e.g., `WRITE`, `CLWB` and `SFENCE`) in both the pre- and post-failure stages for each injected failure. During testing, it first replays the pre-failure trace and updates a shadow PM that keeps track of the status of PM locations. Then, it replays the corresponding post-failure trace to identify any reads to inconsistent locations.

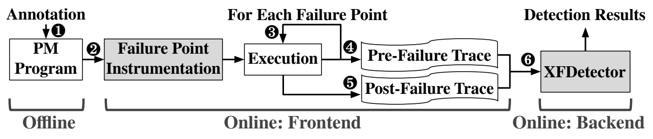


Figure 3. An overview of XFDetector.

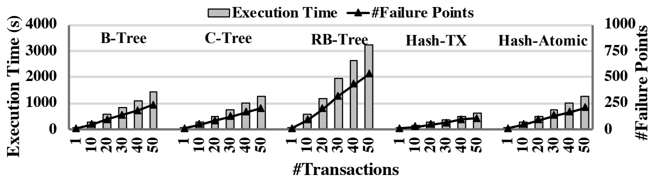


Figure 4. The execution time of micro benchmarks with variable numbers of pre-failure transactions.

Failure Injection. XFDetector's detection method requires the program to crash and resume execution afterward, such that it can trace the pre- and post-failure stages and perform detection. To cover all interactions across failure, the naive but costly solution is to inject failure points for *all possible* interleavings of PM updates. We observe that updates to PM are not guaranteed to persist until explicitly written back (e.g., using a `persist_barrier()`). We refer to such a point that explicitly writes back data to PM as an *ordering point.* As such, persistent data can only transition from an inconsistent state to a consistent state after an ordering point. Therefore, XFDetector only needs to perform cross-failure detection immediately *before* each ordering point.

4 Key Results

We evaluated XFDetector in a real PM system. Workloads include Intel PMDK's persistent data structures and databases. Figure 4 presents the testing time of XFDetector. We scale the number of pre-failure transactions and keep the post-failure procedure constant (one transaction). The primary axis in Figure 4 indicates the execution time (wall-clock time) of detection with variable numbers of pre-failure transactions, and the secondary axis indicates the number of failure points in the pre-failure stage. This experiment shows that the execution time increases *linearly* as the number of failure points increases. We also evaluate the testing capability. XFDetector can detect synthetic bugs that we injected into the workloads, and we found 4 new bugs that have not been detected by prior works, due to the lack of end-to-end cross-failure detection.

References

[1] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *EuroSys*, 2014.
[2] Intel. An introduction to pmemcheck. http://pmem.io/2015/07/17/pmemcheck-basic.html.
[3] Intel. Persistent memory programming. https://pmem.io/.
[4] Intel. Redis. https://github.com/pmem/redis/tree/3.2-nvml, 2018.
[5] Lenovo. Memcached-pmem. https://github.com/lenovo/memcached-pmem, 2018.
[6] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *ASPLOS*, 2020.
[7] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *ASPLOS*, 2019.
[8] Kevin Oleary. How to detect persistent memory programming errors using Intel Inspector - Persistence Inspector, 2018.