# Cross-Failure Bug Detection in Persistent Memory Programs

Sihang Liu
University of Virginia

Korakit Seemakhupt
University of Virginia

Yizhou Wei
University of Virginia

Thomas Wenisch
University of Michigan

Aasheesh Kolli
Penn State University
VMware Research

Samira Khan
University of Virginia

## Abstract

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. 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.

However, maintaining crash consistency is not trivial. We identify that a consistent recovery critically depends not only on the execution before the failure, but also on the recovery and resumption after failure. We refer to these stages as the pre- and post-failure execution stages. In order to holistically detect crash consistency bugs, we categorize the underlying causes behind inconsistent recovery due to incorrect interactions between the pre- and post-failure execution. First, a program is not crash-consistent if the post-failure stage reads from locations that are not guaranteed to be persisted in all possible access interleavings during the pre-failure stage — a type of programming error that leads to a race that we refer to as a *cross-failure race*. Second, a program is not crash-consistent if the post-failure stage reads persistent data that has been left semantically inconsistent during the pre-failure stage, such as a stale log or uncommitted data. We refer to this type of bugs as a *cross-failure semantic bug*. Together, they form the *cross-failure bugs* in PM programs. In this work, we provide XFDetector, a tool that detects cross-failure bugs by automatically injecting failures into the pre-failure execution, and checking for cross-failure races and semantic bugs in the post-failure continuation. XFDetector has detected four new bugs in three pieces of PM software: one of PMDK's examples, a PM-optimized Redis database, and a PMDK library function.



*ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland*



ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378452

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

***Keywords.*** Persistent Memory, Crash Consistency, Testing, Debugging

**ACM Reference Format:**
Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378452

## 1 Introduction

Persistent memory (PM) technologies feature high performance, byte-addressability, and persistence. PM modules are placed on the memory bus and accessed through a load/store interface, thereby blurring the boundary between memory and storage. As Intel has released Optane DC Persistent Memory [21], this technology has finally become available to the industry, researchers, and developers. The arrival of PM has spurred the development of file systems [14, 33, 34, 44, 64, 68, 71, 72], databases [4, 25, 37, 49], key-value stores [9, 69, 70], and custom programs that directly manage persistent data [8, 12, 20, 57, 73]. All of these software systems are expected to recover to a consistent state and be able to resume execution in the event of a failure (e.g., power failure or system crash). We refer to the ability to restore to a consistent state as the crash consistency guarantee.

Due to the volatile caching and reordering of writes within the memory hierarchy, programs need to carefully manage the order in which a write becomes persistent when implementing a crash-consistent program. For example, in the commonly used undo logging mechanism [11, 23, 31], a log of data values must be written to PM *before* a corresponding update happens. Therefore, PM systems have introduced new instructions (e.g., `CLWB` and `SFENCE` from x86 [26], and `DC CVAP` from ARM [3]) to enforce such an ordering, and further provided higher-level abstractions (e.g., transactional libraries [11, 23, 31, 65]) for better programmability. However, it is still not trivial to guarantee crash consistency, as programmers need to have a good understanding of the hardware primitives and/or the semantics of libraries.

Recent works have implemented testing tools for PM programs and detected bugs due to misuse of low-level primitives and libraries when the program is updating persistent objects [22, 42]. Required by the crash consistency guarantee, that is a program returns to a consistent state and resumes the execution after a failure, a testing tool is expected to detect inconsistencies during the *entire* procedure of execution, recovery, and resumption. As such, only testing the part of the execution before a failure happens is insufficient. In this work, we identify that a crash-consistent program must ensure a correct interaction between the execution stage before and after the failure. Therefore, a program *first* needs to correctly implement certain crash consistency mechanisms (e.g., undo/redo logging [5, 6, 10, 23, 28, 31, 65], checkpointing [30, 60], or shadow paging [13, 56]) to ensure data consistency *before* failure. And *second*, after failure, the associated recovery procedure must properly restore PM to a consistent state. We refer to the stages before and after the failure as the *pre-failure* and *post-failure* stages. The pre- and post-failure stages are required to work collaboratively to guarantee crash consistency. If the interaction between the two stages is incorrect, the program might not recover to a consistent state. In the previous undo logging example, even if the program correctly maintains undo logs during the pre-failure stage, the post-failure execution might still read inconsistent data if the recovery procedure does not correctly roll back incomplete updates according to the undo logs. Hence, both the pre- and post-failure execution stages are critical to the crash consistency guarantee. In this work, we seek to test the crash consistency guarantee holistically, considering both the pre- and post-failure execution stages.

In order to holistically detect crash consistency bugs, we first need to precisely define the incorrect interactions between the pre- and post-failure execution stages. In this work, we categorize such interactions into two classes: (1) cross-failure race, and (2) cross-failure semantic bug. Next, we explain both scenarios in detail.

The most common incorrect interaction is that the post-failure execution may read data that is not guaranteed to have persisted in all possible interleavings during the pre-failure stage. Analogous to data races in multithreaded programs, the post-failure execution acts as a "thread" that executes "concurrently" with the pre-failure execution. Without properly orchestrating the "concurrent execution" by enforcing the persistence and the ordering of writes to PM, the post-failure execution might read from locations that were not persisted before the failure. We refer to this scenario as a *cross-failure race*. However, not every cross-failure race leads to a crash consistency issue. Instead, much like races on synchronization primitives that are inherent, cross-failure races are sometimes necessary to enable a correct recovery. For example, suppose the validity of an undo log is indicated by a valid bit. During the post-failure execution, the recovery code must read this valid bit to check whether the undo log needs to be applied to overwrite a potentially inconsistent location. The pre-failure write that sets the valid bit inherently races with the post-failure read, but the recovery outcome is well defined for all possible scenarios of the race. We refer to such intentional races as *benign cross-failure races*, as they do not lead to crash consistency issues.

Even in the absence of cross-failure races, the program can still be semantically incorrect and cause inconsistencies across the failure. For example, under the checkpointing-based recovery mechanism, the post-failure execution should read only from data in the most recent committed checkpoint. Data in earlier checkpoints have been persisted, and accesses to it during recovery do not race, yet these data differ from the latest checkpoint. As such, reading from older checkpoints during the post-failure stage violates the semantics of the crash consistency mechanism. Similar to the cross-failure race, this buggy scenario can only be detected in the event of a failure. However, the difference is a cross-failure race returns a non-deterministic outcome but such a scenario is always buggy if the program fails at a certain point. Therefore, we refer to the second type of incorrect interaction as a *cross-failure semantic bug*.

We collectively refer to these two classes of programming errors as cross-failure bugs. In both cases, the program reads from PM locations that are regarded as *inconsistent*, either because the update to the location is not guaranteed to be persisted before failure, or it is treated as invalid by the semantics of the crash consistency mechanism. The goal of this work is to build upon our definitions of the cross-failure bugs to provide a tool that automatically detects these bugs and validates a PM program's crash consistency guarantee. We propose XFDetector (Xross-Failure Detector) that detects inconsistencies across both the pre- and post-failure stages. At the high-level, XFDetector takes two steps in detection. First, at runtime, XFDetector traces PM operations in both the pre- and post-failure stages. Second, XFDetector replays the two traces and updates a shadow PM to reflect the status of each PM location based on the operations in the trace, such as whether updates have been persisted and data is semantically consistent. The status then enables the detection of cross-failure bugs. In order to generate both the pre- and post-failure traces for testing, XFDetector atomically injects failure points into the PM program. Based on our observation that a program can only enter a consistent state after an explicit writeback to PM (e.g., a `CLWB` followed by an `SFENCE`), XFDetector only injects failures to such points to reduce the number of post-failure executions.

The contributions of this work are the following:

- This work shows that the crash consistency guarantee relies on the correct interaction between the pre- and post-failure stage of a PM program.
- We categorize the incorrect cross-failure interactions into two classes: (1) cross-failure race, where the post-failure

execution reads from non-persisted data, and (2) cross-failure semantic bug, where the post-failure execution reads from semantically inconsistent data.

- Based on the categorization and definition, we implement XFDetector[1], a tool that automatically injects failures into programs, and detects cross-failure bugs by replaying traces of the pre- and post-failure stages.
- XFDetector has detected four new bugs in three pieces of PM software: one of PMDK's examples, a PM-optimized Redis [25] database, and a PMDK library function [23].

## 2 Background and Motivation

In this section, we first introduce programming for persistent memory (PM) systems and its difficulties. Then, we discuss the cause of inconsistencies across failure.

### 2.1 Programming for Persistent Memory is Hard

Persistent memory (PM) technologies, such as Intel's Optane DC Persistent Memory [21], allow programs to directly manage persistent data in memory. Without the OS indirection, programs can fully leverage the high performance and persistence of PM systems. On the other hand, the burden of maintaining the consistency of data in PM lies on programs. We refer to the ability of recovering to a consistent state after failure (e.g., power outage or system crash) as the crash consistency guarantee. A crash-consistent program is expected to recover from a failure and resumes its execution as if the failure has never happened. While, due to the processor's caching and buffering in the volatile memory hierarchy and reordering of writes to memory, the order a write becomes persistent can be different from the intended program order. For simplicity, we refer to the act that a write becomes persistent as a *persist*. To guarantee the persistence and the ordering of writes, PM systems provide instructions, such as `CLWB` and `SFENCE` from x86 [26], to ensure a write to PM has been properly persisted and ordered with other persists. We use a `persist_barrier()`to refer to a sequence of "`CLWB`; `SFENCE`" that writes back a selected cache line and orders it before future persists. Built upon these low-level instructions, there are also high-level libraries, such as Intel's PMDK [23], that abstract away the low-level instructions for better programmability. Both levels of support allow programs to enforce the persistence and the correct ordering of writes to PM, and thereby, guarantee crash consistency.

However, it is not trivial to guarantee crash consistency, as programmers need to have a good knowledge of the primitives and/or PM libraries. Prior works have implemented specialized testing tools for PM programs and found bugs due to misuse of these hardware primitives and library functions when the program performs updates to PM [22, 42]. However, only testing the normal execution stage is insufficient as crash consistency has two fundamental requirements: (1) the program needs to correctly follow crash consistency mechanism to ensure data consistency *before* a failure happens, and (2) the recovery code needs to correctly restore the PM status back to a consistent state after a failure and resume the previously preempted execution. For simplicity, we refer to the phase before failure as the *pre-failure* stage, and after failure as the *post-failure* stage. Next, we will show two examples that fail to meet these requirements.

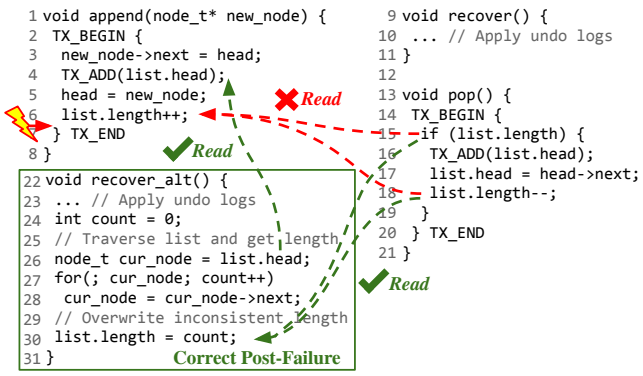


**Figure 1.** An example of an inconsistency in program's post-failure execution.

***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. To guarantee crash consistency, it wraps the updates in a transaction (indicated by `TX_BEGIN` and `TX_END`). Within the transaction, it adds the current PM object to an undo log with a `TX_ADD()` function (line 4), such that if a failure happens in the middle of the transaction, the recovery program can roll back the logs and restore to a consistent state. 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 been persisted. Whether or not this inconsistent `length` can lead to a bug depends on the post-failure execution.

In the naive implementation, the program executes the following steps after the failure: First, it executes the `recover()` function (line 9) that rolls back the incomplete transaction with undo logs. Second, it resumes the program's normal execution. Let's assume the next operation on the linked list is `pop()` (line 13-21), which removes the head node and decrements its `length`. As the `length` was not added to the transaction in the pre-failure execution, the resumption execution keeps using the inconsistent value (as indicated by the red arrows). If the linked list was initially empty before calling the `append()` function and the updated `length` (equals to 1) happens to be persisted before the failure, the resumption execution can even have a segmentation fault as the "if" statement at line 15 becomes "true" and tries to remove a node from the empty linked list.

[1]XFDetector is available at https://xfdetector.persistentmemory.org.

To recover the linked list to a consistent state without requiring the logged `length`, `recover_alt()` traverses the linked list and gets the number of nodes (line 26-28) after applying the undo logs. Then, it *overwrites* the `length` with the correct value (line 30), making the variable `length` consistent. 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). And, after executing the `recover_alt()` function, the function `pop()` also accesses a consistent version of `length` that has been overwritten during the recovery (indicated by the green arrows). Note that the update to `length` at line 30 does not need to be covered by a transaction because its value always gets reset during recovery. Compared to adding `length` to the transaction during the pre-failure stage, this fix is more efficient as the recovery procedure only happens once for each failure. Thus, we refer to this example as an inconsistency in the post-failure stage. However, even with a correct implementation of `recover_alt()`, existing works in crash consistency testing [22, 42] can report a false positive as they only check the pre-failure stage.

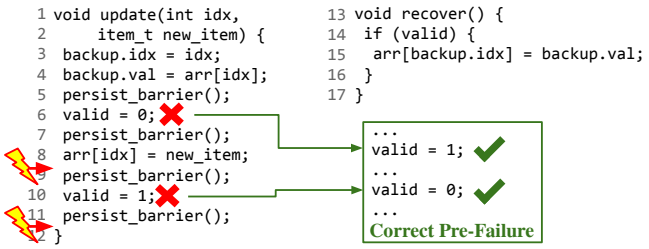


**Figure 2.** An example of an inconsistency in program's pre-failure execution.

***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`). To guarantee crash consistency, the `update()` function first backs up the old data and the updated index (line 3-4). Then, it issues a `persist_barrier()`to writeback the backup and sets a `valid` bit (line 6). After writing back `valid` with another `persist_barrier()`, it performs the in-place update to the array (line 8). And finally, it persists the updates and resets the `valid` bit (line 9-11). Even though this example places a `persist_barrier()`at the correct places, the pre-failure code is still semantically incorrect as `valid` is set to wrong values (corrections are shown in the green box). As a result, the recovery function always performs the wrong operation: If a failure happens before the in-place update has been written back (line 8), the recovery program observes a `valid = 0` and does not roll back the potentially *non-persisted* update. And, if a failure happens after the `update()` function (line 12) has completed, the recovery program rolls back with the *stale* data that is *semantically inconsistent*. Although the bug fix can apply to both pre- and post-failure stages, the more appropriate way is to change the values in the pre-failure stage as the variable `valid` refers to the validity of the backup. For this reason, we refer to this bug as an inconsistency in pre-failure stage. As the consequence of this bug appears after the failure, prior works [22, 42] cannot detect the bug either.

From these two examples, we conclude that it is hard to guarantee crash consistency, not only because PM programming requires a good knowledge of PM low-level instructions and libraries, but also because the pre- and post-failure stages in the program need to work seamlessly. The inability to implement a correct crash consistency mechanism for the pre-failure execution leaves inconsistent data in PM, making it impossible for post-failure execution to restore PM to a consistent state. On the other hand, an incorrect recovery and resumption execution is unable to consistently restore PM. Figure 3 summarizes these two buggy scenarios where the inconsistencies can be due to the pre-failure and/or post-failure execution. Prior works [22, 42] have provided testing tools to detect crash consistency bugs in the pre-failure stage (the shaded area). However, without performing an *end-to-end* test with both stages involved, it is impossible to cover all buggy scenarios.

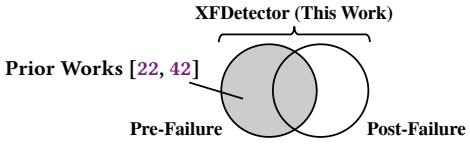


**Figure 3.** Causes of inconsistency.

## 2.2 Causes of Inconsistency

We categorize the incorrect interactions between the pre- and post-failure execution into two classes. The first class of bugs happens when the post-failure execution reads from data that may have not been persisted before the failure. Prior works have suggested that there is a similarity between multithreaded programs and the recovery in certain crash-consistent programs. Lucia et al. model intermittent computing in energy-harvesting devices as concurrency [45, 46]. Chakrabarti et al. make an analogy between races in multithreaded programs and buggy scenarios in their failure-atomic programming model [5]. We further generalize this interaction in PM programs — the execution *before* and *after* a failure can be modeled as a *writer* and a *reader* from two concurrent threads. In the conventional data race, a race happens when at least one of the concurrent accesses to the same memory location is a write [35]. In PM programs, although the pre- and post-failure execution cannot perform real concurrent accesses as they happen in different times, this contentious interaction is still similar to a data race as the value returned by the read after a failure is indeterminate, depending on when the failure happens. Therefore, such a

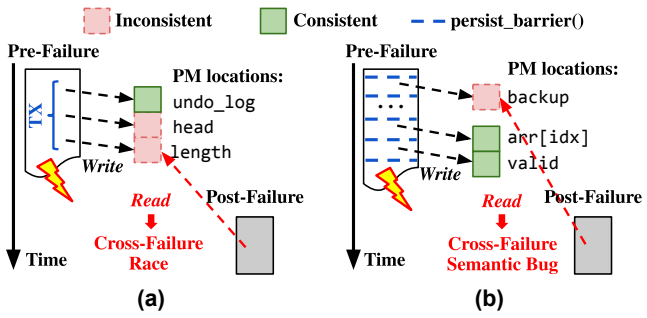


**Figure 4.** Cross-failure bugs from the example of (a) Figure 1 and (b) Figure 2.

read from a potentially non-persisted location may cause undefined behaviors afterward. We refer to reading data that is not guaranteed to be persisted in the post-failure stage as a *cross-failure race*. Figure 4a illustrates the cross-failure race (indicated by the red arrow) between the pre- and post-failure stages in the example of Figure 1. Due to a post-failure bug, the program fails to overwrite the potentially non-persisted `length` modified by the pre-failure "writer", and thus, the post-failure "reader" can access a non-deterministic value.

The second class of bugs happens when the post-failure program reads semantically inconsistent data. Different from the cross-failure race, where the persistence of data is unknown, this type of cross-failure interaction is *always* incorrect as the actual implementation violates the semantics of the intended crash consistency mechanism. Therefore, we name the act of reading data that is semantically inconsistent during the post-failure stage as a *cross-failure semantic bug*. Figure 4b shows the cross-failure semantic bug in the example of Figure 2. Due to the pre-failure bug that incorrectly sets the values of `valid`, the post-failure recovery program reads from a semantically inconsistent `backup`. Because the `valid` bit is incorrectly set by the program implementation, the status after the recovery is always incorrect.

Together, we refer to these two classes of programming errors as *cross-failure bugs*. We refer to data on a PM location as *inconsistent* if it contains updates that are not guaranteed to be written back before a failure, and/or it is semantically inconsistent according to the crash consistency mechanism. A cross-failure bug happens due to the post-failure stage reading data from such inconsistent PM locations that are modified during the pre-failure stage. The *goal* of this work is to detect cross-failure bugs in PM programs by considering both the pre- and post-failure stages holistically.

## 3 Cross-Failure Bugs

In order to detect both types of cross-failure bugs, we first need to precisely define the buggy scenarios. Therefore, in this section, we provide definitions for the cross-failure race and the cross-failure semantic bug.

### 3.1 Cross-Failure Race

***Definition:*** *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 incompletely updated data, leading to inconsistencies after failure. Reading the variable `length` during post-failure recovery in Figure 1 is a typical 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. To formalize the cross-failure race, we first define the following notations:

- $W_x$: A write to the PM location $x$.
- $R_x$: A read from the PM location $x$.
- $M_x$: A read/write from/to the PM location $x$.
- $F$: A failure point that preempts execution.

We then define the following ordering notations:

- $M_x <_{hb} F$: $M_x$ happens before the failure $F$.
- $W_x \leq_p W_y$: $W_y$ may not persist before $W_x$ is persisted.
- $W_x \leq_p F$: $W_x$ has been persisted before the failure $F$.

Therefore, we define a pre-failure write $W_x$ as: $W_x <_{hb} F$, and a post-failure read $R_x$ as $F <_{hb} R_x$. A read $R_x$ has a cross-failure race with $W_x$ iff:

$$W_x <_{hb} F \bigwedge F <_{hb} R_x \bigwedge \neg \left(W_x \leq_p F\right). \quad (1)$$

In other words, if a write is not guaranteed to be persisted before the failure, reading its location during the post-failure execution can cause a cross-failure race. Next, we introduce a special case of the cross-failure race that does not lead to inconsistencies but is necessary for recovery.

***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 non-persisted 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 the *benign cross-failure race*. For example, reading the valid bit of undo logs during the post-failure recovery is regarded as a benign race, as the valid bit enables the recovery program to determine which version is consistent. The checksum-based recovery mechanism (last row in Table 1) is another example of the benign cross-failure race, as the post-failure recovery needs to read potentially non-persisted data and its associated checksum to verify data consistency. In these scenarios, a write to such location inherently races with the post-failure read, while the outcome is always well defined and thus, does not cause any inconsistency. Benign cross-failure races are typically used to determine the consistency status of other PM objects.

| Mechanism | Description | Data Consistency |
|---|---|---|
| Undo logging [11, 18, 23, 39, 43] | Keeps a backup of the old data before performing the in-place update. If a failure happens during the transaction, the recovery mechanism reverts the update with the backup. | If the transaction has been committed, the updated data is consistent. Otherwise, the log is consistent. |
| Redo logging [16, 65, 67] | Performs updates to the log instead of updating in place. If a failure happens during the transaction, the recovery mechanism discards the incomplete redo log. | If the redo log has not been committed, the existing data is consistent. Otherwise, the committed log is consistent. |
| Checkpointing [17, 30, 60] | Creates a checkpoint (i.e., snapshot) of persistent data periodically. After a failure, the recovery mechanism reverts to the last committed checkpoint. | Data in the latest committed checkpoint is consistent. |
| Shadow paging [19, 38, 56] | Performs copy-on-write such that data under modification has a separate copy. Once all updates to the shadow object are completed, the mechanism swaps the original data with the shadow object (e.g., by atomically updating a persistent pointer). | If the shadow object has been committed, data in the shadow object is consistent. Otherwise, the old data is consistent. |
| Operational logging [48, 51] | Logs operations instead of data. If a failure happens during the operation, the recovery mechanism re-executes the logged operation to overwrite the incomplete operation. | Logged operations are consistent. |
| Checksum-based recovery [33, 59, 65] | Determines the consistency status of the modified data using checksums. If a failure happens, the recovery program first reads the data in place and then uses its checksum to determine the consistency. | Data protected by the corresponding checksum is consistent. |

**Table 1.** Data consistency requirements in different crash consistency mechanisms.

### 3.2 Cross-Failure Semantic Bug

***Definition:*** *The post-failure execution reads from data updated during the pre-failure stage that is semantically inconsistent according to the program.*

The second type of cross-failure bug covers inconsistencies defined by the program semantics. PM programs typically follow certain crash consistency mechanisms. Even if a PM location is persisted before failure, it can still be semantically inconsistent if it violates the corresponding data consistency requirements. Table 1 lists the data consistency requirements of common crash consistency mechanisms. Among these different mechanisms, we identify that most crash consistency mechanisms 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. Whereas, the inconsistent version should be discarded or overwritten. These mechanisms typically use a commit variable to indicate whether a set of PM addresses belongs to a consistent version. Data in a set of PM addresses are regarded as consistent only if they were updated between the last two updates to the associated commit variable. For example, in the undo logging mechanism, the program first logs the original data and sets the commit variable (a valid bit) of the log. Then, it performs the in-place update and unsets the commit variable. If a failure happens after the last update to the commit variable, then the in-place update is the consistent version, as it was modified between the last two updates to the commit variable.

We formalize this commonly used version-based crash consistency mechanism by introducing some extra notations:

- $C_{x_i}$: The $i$-th write to the PM address $x$ that alters the consistency status of other PM addresses. We refer to the write as a *commit write* and variable on $x$ as a *commit variable.*
- $S_x$: A set of PM addresses, i.e., $\{m_1 ... m_n\}$, associated with the commit variable on $x$.

In programs that consist of more than one commit variable, their associated PM address sets need to be disjoint, i.e., given two commit variables on address $x$ and $y$, then

$$S_x \cap S_y = \varnothing. \tag{2}$$

Let the last commit write be the $n$-th write to $x$, i.e., $C_{x_n}$, The PM addresses in $S$ are semantically consistent iff:

$$\forall m_i \in S_x, C_{x_{n-1}} \preceq_p W_{m_i} \bigwedge W_{m_i} \preceq_p C_{x_n}. \tag{3}$$

### 3.3 Summary

The Venn diagram in Figure 5 summarizes the two classes of cross-failure bugs. The first class of cross-failure bug is the cross-failure race that reads data not guaranteed to be persisted before a failure, unless it is an intended benign cross-failure race. The second class is the cross-failure semantic bug that reads semantically inconsistent data. As the focus of this work is to detect crash consistency bugs due to cross-failure interactions, we do not consider other types of bugs. Next, we describe our key ideas for detection based on the definition of these cross-failure bugs.

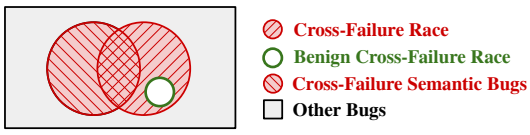


**Figure 5.** Two classes of cross-failure bugs.

# 4 Key Ideas of XFDetector

So far, we have described the definitions of the cross-failure bug. It would be greatly helpful to programmers if there is a way to detect them. In this work, we propose XFDetector, a Xross-Failure Detector. At the high-level, XFDetector traces PM operations in both the pre- and post-failure stages, and then detects inconsistencies due to buggy interactions between these two stages. In the design and implementation of XFDetector, we answer two research questions: (1) What is a proper approach to determine data consistency in order to detect cross-failure races and semantic bugs? (2) What is an efficient way to inject failures into the program to cover all cross-failure interactions?

## 4.1 Data Consistency

***Challenge.*** Detecting inconsistencies across the failure requires determining whether data read by the post-failure execution is consistent. However, data consistency is not self-contained by data but depends on program's manipulation of persistent data. The challenge is to determine data consistency based on the program execution.

***Solution.*** The consistency status of persistent data changes as the program performs updates to PM. Therefore, to capture the updates, XFDetector traces PM operations (e.g., `WRITE`, `CLWB` and `SFENCE`) in the pre- and post-failure execution stages. To detect cross-failure bugs, XFDetector implements a shadow PM that records the status of each PM location. XFDetector first replays the pre-failure trace and then the corresponding post-failure trace. XFDetector updates the status of the shadow PM while replaying the traces, and checks if the post-failure accesses satisfy the conditions described in Section 3.

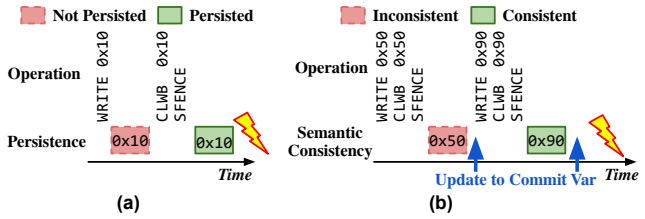


**Figure 6.** Examples of detecting (a) a cross-failure race and (b) a cross-failure semantic bug based on the data consistency status of PM locations.

Figure 6a shows an example of detecting a cross-failure race based on the persistence of data. A PM location, `0x10`, first gets modified and then the persistence status becomes *not persisted* as this update is not guaranteed to be written back. Then, a sequence of `CLWB` and `SFENCE` writes back this location and thus, changing the status to *persisted*. Figure 6b shows another example that detects cross-failure semantic bugs by determining the data consistency status according to the updates to the *commit variable* (indicated by the blue arrows). There are two updates to the locations `0x50` and `0x90` that have been persisted before the failure. However, being persistent does not mean the locations are consistent. As the location `0x90` is last modified between the last two updates to the commit variable, it is regarded as semantically consistent, while the other location `0x50` is not.

## 4.2 Failure Injection

***Challenge.*** XFDetector needs to inject failures during the program execution in order to trigger both the pre- and post-failure stages. We refer to such injected failures as *failure points*. To capture all incorrect cross-failure interactions, the naive solution is to inject failure points for all possible interleavings of PM updates, considering the PM status can change after each update. However, this exhaustive method is extremely costly as XFDetector needs to perform post-failure execution for every failure point.

***Solution.*** We observe that updates to PM are not guaranteed to be persisted until explicitly written back (e.g., using a `persist_barrier()`). We refer to a point in the program that explicitly writes back data to PM before any future PM operations as an *ordering point*. As such, persistent data can only transition from an inconsistent state to a consistent state after an ordering point. Therefore, it is only necessary to check the consistency status immediately *before* each ordering point. Based on this observation, XFDetector only injects failure points before each ordering point[2]. The ordering points that XFDetector concerns about include both low-level operations (e.g., `SFENCE`) and high-level functions that enforce writeback (e.g., `TX_ADD()` in PMDK [23]).

# 5 Implementation of XFDetector

## 5.1 An Overview of XFDetector

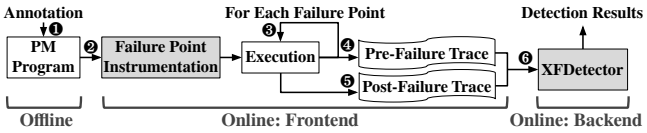


**Figure 7.** An overview of XFDetector.

Figure 7 shows an overview of XFDetector's detection procedure that consists of three steps: (1) an *offline* step that requires annotation of the region-of-interest (RoI) in both the pre- and post-failure stages, (2) an *online frontend* that injects failure points and generates traces, and (3) an *online backend* that detects and reports cross-failure bugs based on the traces. The following is an overview of the detection procedure: First, the programmer annotates the source code and compiles it with XFDetector library (step ❶). Second, XFDetector automatically instruments the program with

[2]Checksum-based mechanism is an exception that data consistency relies on the verification of the checksum. We briefly discuss how to inject additional failure points for this mechanism in Section 5.5.

failure points before its execution begins (step ❷). During execution, it follows a procedure of execute pre-failure stage – suspend at the failure point – execute the corresponding post-failure stage, until it completes or reaches the termination point (step ❸). During execution, it generates both the pre-failure (step ❹) and post-failure traces (step ❺). Finally, as the frontend is tracing, the backend performs detection and reports the detection results (step ❻).

### 5.2 Software Interface

XFDetector provides a C/C++-compatible interface as listed in Table 2. XFDetector has two types of functions. The first type controls the detection procedure and allows programmers to select the region-of-interest (RoI) for detection. The second type is used for annotating the source code to support detection. For trusted code (e.g., implementation of library functions), programmers can choose to skip the injection of failure points and bug detection. Programmers can also add additional failure points on demand. To expose crash consistency semantics in programs directly built on low-level primitives, XFDetector allows programmers to register the commit variable and its associated PM objects. By default, if there is only one commit variable and no object is specified, it covers all PM locations. During the execution of XFDetector, reads from the selected commit variables are marked as benign cross-failure races, without being reported as bugs. Both types of functions take two arguments, `condition` and `stage`, which allow programmers to manage when the function takes effect. It is worth pointing out that programmers only need to use the functions to select the region for detection, without any need for additional annotation when testing programs that are built on top of PM libraries. The functions for annotation are needed only when testing programs that directly use low-level primitives or the implementation of PM libraries.

| | Function | Description |
|---|---|---|
| Control | RoIBegin(condition, stage) | Mark a region for XFDetector detection |
| | RoIEnd(condition, stage) | |
| | completeDetection(condition, stage) | Terminate detection |
| Detection Annot. | skipFailureBegin(condition) | Mark a region that skips failure points |
| | skipFailureEnd(condition) | |
| | addFailurePoint(condition) | Add additional failures |
| | skipDetectionBegin(condition, stage) | Mark a region that skips detection |
| | skipDetectionEnd(condition, stage) | |
| | addCommitVar(variable) | Mark a commit variable and associated address |
| | addCommitRange(variable, addr, size) | |

**Table 2.** XFDetector software interface.

### 5.3 Tracing Mechanism

XFDetector's tracing mechanism generates a trace of low-level instructions, including PM reads and writes, fences, and writeback operations. XFDetector leverages PMDK's address derandomization option to map PM locations to a predefined virtual address range to distinguish PM operations, and keeps the virtual address of each PM object the same across different executions to simplify the detection process (by setting the `PMEM_MMAP_HINT` environment variable [24]). Due to the high performance overhead from tracing at such fine granularity, XFDetector optimizes the tracing procedure for programs built on PMDK [23] by skipping the trace of internal implementation but only maintaining a trace of library function calls, such as PM transactions and allocations. This way, the user code is traced at instruction granularity and internal library code is traced at function granularity. In the trace entry, XFDetector keeps track of the operation's instruction pointer, and the source/destination addresses and their sizes. The instruction pointer is used for backtracing the bug, and the address and size differentiates PM objects.

### 5.4 Detection Procedure

The detection procedure in XFDetector consists of two parts: a frontend that injects failures and traces PM operations, and a backend that detects bugs based on the traces.

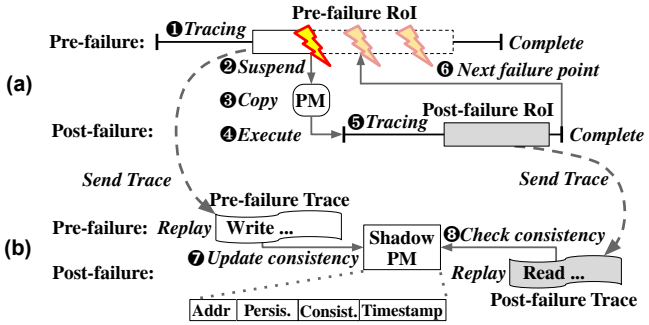


**Figure 8.** XFDetector's (a) frontend and (b) backend.

***Frontend.*** We implement the frontend based on Intel's Pin [47] to perform tracing and failure injection. In order to inject failures for testing, before executing the program, the frontend first locates all ordering points in the binary and then instruments the binary with failure handlers before each ordering point (as described in Section 4). After instrumentation, the frontend performs tracing and failure injection during execution (as shown in Figure 8a). In the *pre-failure stage*, XFDetector collects a trace of PM writes and library functions (step ❶). When encountering a failure point (i.e., failure handler) within the RoI, XFDetector suspends the program (step ❷), makes a copy of the current PM image (a pool file on PM) (step ❸)[3], and spawns its post-failure execution (step ❹). Then, in the *post-failure stage*, XFDetector generates another trace (step ❺) until it reaches the annotated termination point (or naturally terminates). Then it continues the pre-failure execution and moves

[3]The copy of PM image contains all updates (including those not persisted before the failure point). XFDetector maintains a shadow PM to track which locations have been persisted for the purpose of detection.

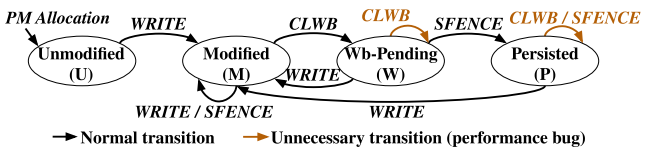


**Figure 9.** Transitions of the persistence state.

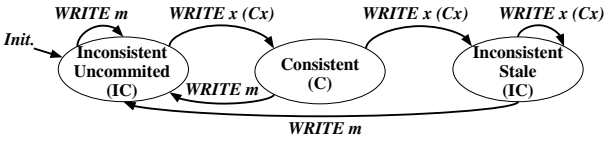


**Figure 10.** Transitions of the consistency state.

on to the next failure point (step ❻). During the remaining pre-failure execution, XFDetector incrementally traces new operations instead of starting over from the beginning for better performance. While tracing, the frontend sends the already completed trace to the backend (through a pre-failure trace FIFO and a post-failure trace FIFO) for detection. This way, the detection procedure can overlap with tracing. Next, we describe the backend detection mechanism.

***Backend.*** XFDetector maintains a *shadow PM* with the following fields for each PM address to record their status: (1) a persistence state field that can be *unmodified (U)*, *modified (M)*, *writeback-pending (W)*, and *persisted (P)*, (2) a consistency state field that can be *consistent (C)* or *inconsistent (IC)* according to program semantics, and (3) a *timestamp* $T_{last}$ that indicates the last time the address was modified. XFDetector uses the *PM state* field to detect the cross-failure race, the *consistency state* field to detect the cross-failure semantic bug, and the *timestamp* to update the consistency state based on the commit variable. Each commit variable keeps another *timestamp* $T_{prelast}$ that indicates the pre-last time it was modified. Each timestamp is obtained from a *global timestamp* that increments after each ordering point.

During the detection procedure, XFDetector replays the traces in the order of pre-failure and post-failure:

*Pre-failure Trace:* XFDetector's backend replays the pre-failure trace by updating the shadow PM for each write and each library function that modifies PM (step ❼). XFDetector updates both the *persistence state* and *consistency state* according to the operations in the trace. For the persistence state of a PM location, XFDetector follows the finite-state machine in Figure 9. In brief, a `WRITE` changes the state to *modified*, a `CLWB` changes the modified state to *writeback-pending*, and finally, an `SFENCE` changes the state to *persisted*[4]. For consistency state, this work supports common crash consistency mechanisms that use commit variables and those that are built on PMDK transactional functions[5]. Figure 10 shows the consistency state transition of a PM location $m$ according to the updates to its associated commit variable $x$ ($C_x$ refers to a write to the commit variable), where the location $m$ can be *inconsistent* in two ways: being uncommitted or stale. In brief, an uncommitted location becomes consistent after an update to the commit variable, and a stale location first gets updated and then becomes consistent after being committed. XFDetector handles PMDK transactional functions in a similar way to PMTest [42], where objects that have been added to the transaction are regarded as consistent. During the update of the shadow PM, XFDetector also reports performance bugs that use unnecessary PM operations (e.g., redundant writebacks as indicated by the yellow edges in Figure 9), and unnecessary library functions (e.g., duplicated `TX_ADD()` functions for the same PM object).

*Post-failure Trace:* XFDetector then replays the corresponding post-failure trace (step ❽). Different from processing the pre-failure trace, writes in post-failure change the consistency status to *consistent* as they *overwrite* the old data. Inconsistencies introduced by these writes will be tested later when this code region runs as the pre-failure stage. For each read, XFDetector first checks the *consistency state* and then the *persistence state* of the target location, as reading a consistent location is certainly bug-free, while reading a persisted location can still be semantically inconsistent. Reading a commit variable is a *benign cross-failure race* and not regarded as a bug. On detection of a cross-failure bug, XFDetector reports the file name and the line number of the reader and the last writer that cause the bug. Next, we illustrate the detection procedure with an example.

***Example.*** Figure 11 demonstrates the detection procedure, where Figure 11a, 11b, and 11c show the pre- and post-failure trace, the shadow PM, and the code, respectively. The `valid` variable is marked as a commit variable because it decides the validity of the backup and the in-place update.

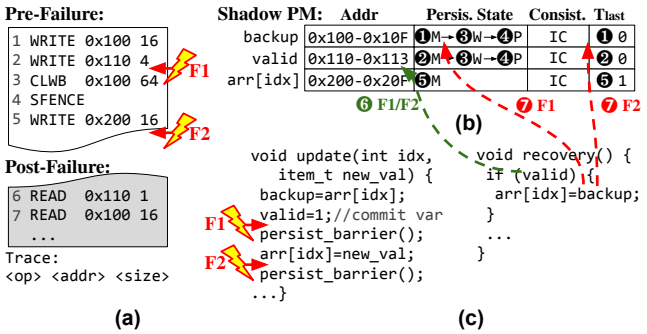


**Figure 11.** (a) The pre- and post-failure traces, (b) the states in the shadow PM, and (c) the code demonstrating the steps of the detection procedure.

[4]XFDetector also handles non-temporal writes and other types of fence.
[5]We reserve an extensibility as discussed in Section 5.5 to support other crash consistency mechanisms.

XFDetector injects a failure point before each of the two ordering points (F1 and F2 before each `persist_barrier()`), and each failure point triggers their corresponding post-failure execution. We take the first two entries (line 6 and 7) from the post-failure trace for demonstration. Initially, the *global timestamp* is 0 and all PM addresses in the example are *unmodified*. Next, we demonstrate the detection step-by-step. Line 1: creates `backup` and updates its PM status to *modified*. Line 2: sets `valid` and updates its PM status to *modified*. As there is no update before the commit timestamp, XFDetector does not change the consistency state of any PM location. F1: the first failure triggers the post-failure execution. Line 6 (F1): reads from `valid` (the commit variable). Line 7 (F1): reads from `backup` for rolling back. However, as the PM state of `backup` is *modified*, XFDetector reports a cross-failure race. Then, it continues pre-failure execution from F1. Line 3: writes back a cache line that contains both `backup` and `valid`, and updates both PM status to *writeback-pending*. Line 4: places an `SFENCE` to make sure previous pending writebacks are complete, and increments the *global timestamp*. Line 5: updates the variable `arr` in-place and the persistence status becomes *modified*. F2: the second failure triggers the post-failure execution. Line 6 (F2): reads from `valid` (the commit variable). Line 7 (F2): reads from `backup` for rolling back. However, as the consistency state is *inconsistent*, XFDetector reports a cross-failure semantic bug. This bug is due to `backup` not being updated *before* the last update to the commit variable (`valid`). In summary, XFDetector reports a cross-failure race at the first failure point (F1), and a cross-failure semantic bug at the second failure point (F2).

***Optimizations.*** XFDetector takes the following optimization strategies for better efficiency without degrading its detection capability. (1) *Eliminate unnecessary consistency checks*: In the post-failure stage, there can be multiple reads from the same PM location that was modified during the pre-failure stage. XFDetector only checks the first read and skips the rest as the result would be the same. (2) *Eliminate unnecessary failure points*: In the pre-failure stage, there can be two ordering points without any PM operations in between (e.g., two consecutive calls to PM library functions with ordering points). XFDetector does not inject a failure point in the middle for better performance.

***Complexity.*** Assuming there are $F$ failure points in the pre-failure stage, and each corresponding post-failure execution has $P$ operations on average, the complexity of the detection procedure is $O(F \cdot P)$. We observe that the post-failure execution in most crash consistency mechanisms takes a small, constant number of steps to recover from the failure. For example, an undo logging mechanism only recovers the last incomplete transaction. Therefore, the detection time scales *linearly* with the number of failure points in most scenarios. We evaluate XFDetector's scalability in Section 6.2.2.

### 5.5 Extensibility

This section describes the extensibility of XFDetector to support other PM systems and detect other types of bugs.

***Extending Operation Tracing.*** XFDetector decouples the frontend tracing from the backend detection. The frontend of XFDetector is built on Intel's Pin [47] for fine-grained, automated tracing. Although Pin is limited to user-space programs and Intel processors, the backend of XFDetector can be attached to other tracing frameworks, such as the software-directed tracing in WHISPER [53] and PMTest [42].

***Extending Detection Capability.*** We summarize the possible approaches for extending XFDetector as the following points. First, XFDetector functions (Table 2) can work as building blocks to support other PM libraries. Take our implementation as an example, we skip the detection of PMDK's internal transactions but instead explicitly add a failure point for each library function that contains ordering points. This way, XFDetector only needs to handle programmer's code. Second, if the target program applies a crash consistency mechanism that does not follow the approach described in Section 3.2, programmers may need to modify the tool and provide extra annotations. For example, to support a version-based mechanism that does not take the latest copy but uses a specific one in the log, programmers need to add extra timestamps to track when the log was committed. The checksum-based mechanism is another example, where the consistency status is not determined by a commit variable but uses a pair of data and its associated checksum. To test the correctness of the checksum implementation, programmers may manually place a failure point using XFDetector's library function or modify the failure injection mechanism to automatically add more failure points between ordering points. Third, if a cross-failure bug is beyond the capability of XFDetector, the failure injection framework can work in cooperation with conventional debugging techniques. For example, bugs that depend on data values, such as creating a log using incorrect data, cannot be detected because XFDetector does not track data values. To detect such bugs, programmers may place assertions to check data values in the post-failure code and then use XFDetector's failure injection mechanism to trigger the post-failure execution.

## 6 Evaluation

### 6.1 Methodology

| | |
|---|---|
| CPU | Intel Xeon Gold 6230, 2.1GHz, 20 cores |
| PM | 2×128GB Intel DCPMM, App Direct, Interleaved |
| DRAM | 4×16GB DDR4, 2666MT/s |
| OS | Ubuntu 18.04, Linux kernel 4.15 |
| Tools & Libs | gcc/g++-7.4, Pin-3.10, PMDK-1.6, ndctl-61.2 |

**Table 3.** The evaluated system.

We evaluate our tool, XFDetector in a real system (Table 3) with Intel's Optane DC Persistent Memory Module

(DCPMM). PM is mounted with the DAX option to bypass OS indirections [27]. Table 4 lists the evaluated PM programs, including 5 micro benchmarks from PMDK [23] examples and 2 real-world workloads: Redis [25] and Memcached [37]. The transaction-based programs are built with PMDK's libpmemobj, and the low-level ones are built with libpmem. We annotate the source code with XFDetector interface for cross-failure bug detection. Table 4 lists the lines of code (LOC) of the original version and our annotation. We modify the `Makefile` to link the test program with the shared object of XFDetector's interface for all workloads. We mark the entire program as RoI (both pre- and post-failure) for the micro benchmarks, and select the code region that performs updates to PM objects as the pre-failure RoI and the region that performs recovery as the post-failure RoI for larger real-world workloads.

| | | | Lines of code (LOC) | |
|---|---|---|---|---|
| | Name | Type | Original | Annotation |
| Microbench | B-Tree | Transaction | 981 | 4 |
| | C-Tree | | 698 | 4 |
| | RB-Tree | | 855 | 4 |
| | Hashmap-TX | | 741 | 4 |
| | Hashmap-Atomic | Low-level | 837 | 5 |
| Real | Memcached | | 23k | 10 |
| | Redis | Transaction | 66k | 6 |

**Table 4.** The evaluated PM programs.

## 6.2 Performance

**6.2.1 Execution Time.** This experiment evaluates the execution time of XFDetector. We run each workload with one transaction/query that performs an insertion, and another one for each failure point. Figure 12a shows the wall-clock time (seconds) of XFDetector for each workload. XFDetector takes an average of 40.6 seconds to analyze one insertion operation. We further break down the execution time into two parts: the pre- and post-failure stages. We observe that the post-failure takes the majority of the execution time as XFDetector spawns the post-failure execution for each failure point. We further compare the execution time of XFDetector with a "Pure Pin" configuration where the Pintool only traces the PM read/write operations, and the original program that runs without any tool (Figure 12b). On average

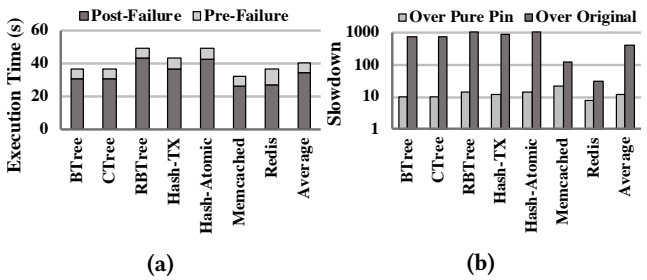


**Figure 12.** Performance of XFDetector: (a) wall-clock time and (b) slowdown over pure Pin and original program.

(Geo. mean), XFDetector is 12.3× slower than "Pure Pin" and 400.8× slower than the original program. We conclude that the repeated post-failure execution is the major bottleneck, and Pintool is the secondary bottleneck. However, the post-failure executions are independent as they operate on a copy of the original PM image, and therefore, can be parallelized. We leave the parallelized detection as a future work.

**6.2.2 Scalability.** This experiment scales the number of transactions performed in the pre-failure stage during detection. As real-world workloads execute upon query, we scale the number of pre-failure transactions in micro benchmarks and keep the post-failure constant (one transaction). The primary axis in Figure 13 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.

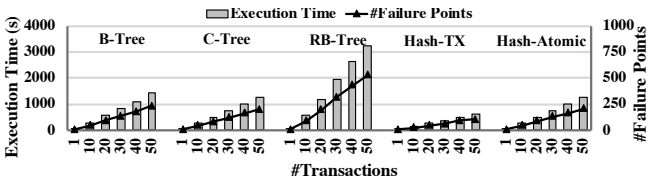


**Figure 13.** The execution time of micro benchmarks with variable numbers of pre-failure transactions.

## 6.3 Detection Capability

In this section, we first validate the debugging capability of XFDetector, and then demonstrate the new bugs we found.

| | PMTest Bug Suite | | | Additional | |
|---|---|---|---|---|---|
| Name | R | S | P | R | S |
| B-Tree | 8 | N/A | 2 | 4 | / |
| C-Tree | 5 | | 1 | 1 | |
| RB-Tree | 7 | | 1 | 1 | |
| Hashmap-TX | 6 | | 1 | 3 | |
| Hashmap-Atomic | 10 | | 2 | 3 | 4 |

**Table 5.** The synthetic bugs for validation (R: cross-failure race, S: cross-failure semantic bug, and P: performance bug).

**6.3.1 Validation.** Table 5 summarizes the synthetic bugs that we have validated using XFDetector. We first validate XFDetector's detection capability with the bug suite from PMTest [42]. As cross-failure semantic bugs are beyond PMTest's scope, we create additional synthetic, cross-failure semantic bugs on top of the Hashmap-Atomic example which is built on low-level primitives. We do not create cross-failure semantics bugs for other workloads as the commit variables are managed by their transactional library functions. We also create other cross-failure race bugs for better validation. The

validation shows that XFDetector is effective in detecting these synthetic bugs and covers more types of bugs than existing works [22, 42].

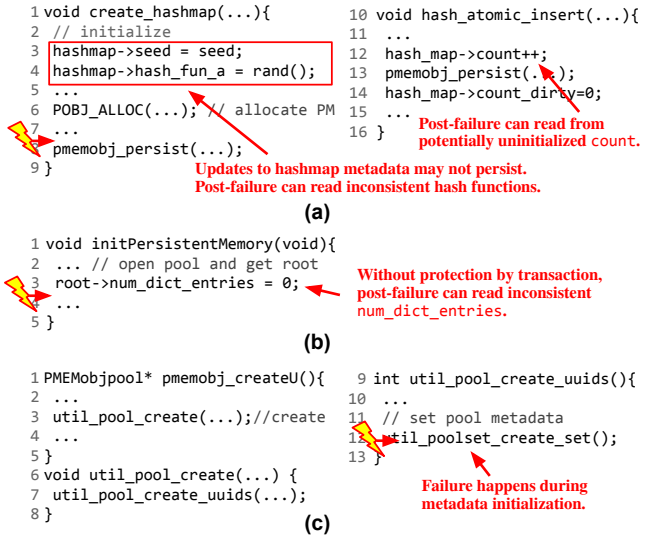


**Figure 14.** New bugs detected by XFDetector in (a) Hashmap-Atomic, (b) Redis, and (c) libpmemobj.

**6.3.2 New Bugs.** XFDetector found new bugs that have not been identified by prior works. **Bug 1** is found in a PMDK example, Hashmap-Atomic (`hashmap_atomic.c:132-138`) that uses the low-level operations to ensure crash consistency. The initialization function (`create_hashmap`) assigns hashing functions and their `seed` as part of the hashmap's metadata (line 3 and 4 in Figure 14a). These updates are not protected by any crash consistency mechanism. Therefore, if a failure happens before they are written back (line 8), the post-failure program can read from invalid function pointers and an invalid seed value that are not completely persisted to PM, leading to a cross-failure race. **Bug 2** is also found in the Hashmap-Atomic example (`hashmap_atomic.c:280`), where the program accesses a potentially uninitialized PM location (`count`). The program allocates a piece of PM when creating the hashmap (line 4 in Figure 14a). If a failure happens right after the allocation, the post-failure program can read the variable `count` (line 12) that may not be initialized. This example happens to use an allocator that implicitly initializes the location with zeros. However, with a different allocator, the implicit initialization is not guaranteed, and therefore, can lead to a cross-failure race as the pre-failure program creates an *unmodified* PM location that is read by the post-failure execution. We only annotated a commit variable, `count_dirty`, to detect these two bugs. **Bug 3** is found in Redis [25] (`server.c:4029`), where the Redis server initializes PM (Figure 14c). Similar to the previous bug, the initialization procedure is not protected by a transaction, and therefore, a failure in the middle of the initialization can lead to a cross-failure race. We did not manually expose any program semantics to detect such bug as Redis is transaction-based. **Bug 4** is found in PMDK's libpmemobj library (`obj.c:1324`). The PM pool creation function, `pmemobj_createU()`, initializes a region of PM and sets its metadata (through `util_pool_create_uuids()`) as demonstrated in Figure 14c. All data have been persisted at the end of the creation function, however, there is no consistency guarantee in the middle. A failure point injected in the middle of the creation process can cause the created PM pool to have incomplete metadata. Then, the post-failure program tries to open the pool for recovery but fails. Although the post-failure `open()` operation is a syscall and out the scope of tracing, XFDetector's failure injection mechanism makes this bug observable. We conclude that XFDetector is effective at detecting cross-failure bugs with minimum annotation.

## 7 Discussion

In this section, we discuss the assumptions and the scope of this work.

***Detection Scope.*** XFDetector can detect cross-failure bugs due to reading non-persisted or semantically inconsistent data. The detection mechanism takes into account the address and the order of PM updates instead of data values (except for commit variables that can affect the procedures in the post-failure stage). Therefore, programming errors such as writing incorrect data values to non-commit variables (e.g., log incorrect data) are out of the scope. Section 5.5 has described the way to extend the capability by incorporating conventional debugging methods with XFDetector.

***Multithreaded PM Programs.*** The frontend of XFDetector is thread-safe by using thread-local storage and Pin's locking primitives, and the backend runs in a separate process without being interfered by the multithreaded workload. Therefore, programmers do not need to adjust XFDetector to test multithreaded programs. The concurrent threads in our workloads perform PM operations on independent tasks (e.g., each thread takes a different request), and therefore, we do not implement cross-failure bug detection for collaborative updates to PM from concurrent threads. However, XFDetector can be extended to support such scenarios by sharing a global timestamp among multiple threads and introduce more program-specific rules for consistency checking.

***External Dependency.*** XFDetector executes the post-failure stage on a temporal copy of the original PM image. Therefore, external events (e.g., I/O) can possibly cause variation among different post-failure executions. However, we did not observe any external events that change the PM status in the evaluated workloads.

## 8 Related Works

***Crash Consistency Mechanisms.*** Prior works have provided a variety of crash consistency mechanisms to ensure the consistency across failure. In general, there are two types of methods. The first type of mechanisms proposes new hardware features for PM system. For example, ATOM [29], Kiln [75], and DudeTM [38] provide efficient hardware transactions, ThyNVM [60] and PiCL [55] propose transparent hardware-based checkpointing, DPO [32] and HOPS [53] introduce new persistency models, and SCA [40], Osiris [74], and Janus [41] provide efficient and crash-consistent PM systems with security guarantees. The second type of mechanisms provides software and library support for existing PM hardware. For example, PMDK [23], NV-Heaps [11], and Mnemosyne [65] provide PM libraries, and PMFS [14], NOVA [71], and BPFS [13] implement PM-optimized file systems to manage persistent data. There are also works that extend multithreading synchronization to the persistence domain, such as Atlas [5], SFR [18], and iDO [39]. XFDetector can be extended with new PM library functions and low-level primitives to detect cross-failure bugs in these PM systems.

***Crash Consistency Testing.*** There have been works that detect inconsistencies in conventional file systems that run on hard drives [7, 15, 50, 52, 63]. Due to the fundamental difference between the hard drive and the byte-addressable PM, these methods are not applicable to PM programs. There are also toolchains specifically designed for PM. PMTest [42] and Pmemcheck [22] test crash consistency in custom PM programs. However, they only consider the pre-failure stage without testing both the pre- and post-failure stages holistically. Therefore, XFDetector has better debugging capability than these works. Yat [36], a tool that validates Intel's PM-optimized POSIX-compliant file system (PMFS [14]), considers both stages by running the recovery code in PMFS and then checking data consistency. However, Yat's approach does not apply to generic programs as it relies on file system check (fsck) to detect inconsistencies. In comparison, XFDetector supports custom PM programs.

***Multithreading and Persistence.*** Prior works have made an analogy between the recovery of PM programs and multithreaded programs. Atlas [5] proposes failure-atomic sections (FASEs) as a crash-consistent programming model and introduces the concept of restart-race-freedom of FASEs that guarantees a correct recovery after a failure. DINO [46] models the intermittent execution in energy-harvesting devices as concurrency and proposes atomic tasks to overcome unexpected behaviors due to "races". The concepts of "race" introduced by these prior works only support their specific programming models. In comparison, this work systematically defines such racing scenarios for generic PM programs and guarantees the correctness of crash consistency mechanisms based on the definitions.

***Conventional Data Race.*** There has been a myriad of works on formalizing [1, 54] and detecting [2, 58, 61, 62, 66] data races in multithreaded programs. The cross-failure race in this work are analogous to the data race, as the value returned by a read is indeterminate and hence the reader may access incorrect data in both scenarios. However, cross-failure races differ in two fundamental ways: (1) the write and the read are separated by a failure without actually being performed concurrently, and (2) the relevant happens-before relations are with the persistence of writes instead of their visibility to other threads.

## 9 Conclusions

In this work, we show that both the pre- and post-failure stages are equally critical to ensure the crash consistency guarantee. We categorize two classes of cross-failure bugs due to incorrect interactions across the failure: the *cross-failure race*, where the post-failure execution reads from a non-persisted data, and the *cross-failure semantic bug*, where the post-failure execution reads from data that violates the crash consistency semantics. We provide XFDetector that detects both classes of cross-failure bugs by holistically considering both the pre- and post-failure stages. XFDetector has detected four new bugs in three pieces of PM softwareg: one of PMDK's examples, a PM-optimized Redis database, and a PMDK library function.

## A Artifact Appendix

### A.1 Abstract

This artifact provides the source code of XFDetector, a testing tool that detects crash consistency bugs in programs for persistent memory (PM) systems. It also provides a set of example workloads and necessary dependencies. At the high-level, out tool, XFDetector detects crash consistency bugs in PM programs by injecting failures during program execution and replying the traces of both the pre- and post-failure execution. As XFDetector is designed for PM systems, this artifact requires real or emulated PM system and a compatible Linux distribution.

### A.2 Artifact Check-list (meta-information)

- **Program:** The testing tool of XFDetector.
- **Data set:** Open-source workloads from Intel and Lenovo.
- **Hardware:** A system with a real or an emulated PM.
- **Output:** Bug reports for test programs.
- **Experiments:** Bug detection and execution time.
- **Publicly available?:** Yes.
- **Code licenses:** BSD.
- **Archive DOI:** https://doi.org/10.5281/zenodo.3619413.

### A.3 Description

**A.3.1 How delivered.** We archived the source code at Zenodo: https://doi.org/10.5281/zenodo.3619413. For the latest version, please check our GitHub page: https://xfdetector.persistentmemory.org.

**A.3.2 Hardware Dependencies.** XFDetector supports systems with a real (e.g., NVDIMM and Intel DCPMM) or an emulated PM. For PM emulation, please see PMDK's documentation for detailed instructions: https://pmem.io/2016/02/22/pm-emulation.html. Note that PM (real or emulated) must be mounted as a DAX file system.

**A.3.3 Software Dependencies.** The following is a list of software dependencies for XFDetector and the test workloads (the listed versions have been tested, other versions might work but not guaranteed).

- OS: Ubuntu 18.04 (kernel 4.15)
- Compiler: g++/gcc-7.4
- Libraries: libboost-1.65 (`libboost-all-dev`), pkg-config (`pkg-config`), ndctl-61.2 (`libndctl-dev`), daxctl-61.2 (`libdaxctl-dev`), autoconf (`autoconf`), and libevent (`libevent-dev`). Other dependent libraries for the workloads are contained in this repository.

**A.3.4 Data Sets.** Our evaluated workload are as follows:

- Five PMDK [23] example workloads (B-Tree, C-Tree, RB-Tree, Hashmap-TX and Hashmap-Atomic).
- Intel's Redis implementation for PM [25].
- Lenovo's Memcached implementation for PM [37].

### A.4 Installation

This artifact is organized as the following structure:

- `xfdetector/`: The source code of our tool.
- `driver/`: The modified driver function for PMDK examples.
- `pmdk/`: Intel's PMDK library, including its examples.
- `redis-nvml/`: A Redis implementation (from Intel) based on PMDK (PMDK was previously named as NVML). This folder will be created after executing the script `init_redis.sh`.
- `memcached-pmem/`: A Memcached implementation (from Lenovo) based on Intel's PMDK library.
- `patch/`: Patches for reproducing bugs and trying our tool.

To build XFDetector and the test workloads, please use the following commands:

```
$ cd <XFDetector Root>
$ export PIN_ROOT=<XFDetector Root>/pin-3.10
$ export PATH=$PATH:$PIN_ROOT
$ make
```

Our tool and test programs also have separate makefiles. Please follow the instructions on our GitHub page if need to build them separately.

### A.5 Experiment Workflow

Figure 7 describes the high-level workflow of XFDetector. The programmer annotates the test program with XFDetector's interface. When the testing begins, the programmer first executes the XFDetector and then executes our Pintool. During execution, our Pintool sends the PM trace entries to XFDetector for testing. We refer to this period as the *pre-failure* stage. Once a trace entry triggers a failure point, our Pintool suspends the pre-failure execution, and lets XFDetector copy the PM image and perform the *post-failure execution* (also uses our Pintool for tracing). The post-failure program, sends a PM trace to XFDetector for cross-failure bug detection during execution. In this artifact, we provide scripts that automate these steps.

### A.6 Evaluation and Expected Result

Before running any program, please execute the following commands under the root directory of XFDetector:

```
$ export PIN_ROOT=<XFDetector Root>/pin-3.10
$ export PATH=$PATH:$PIN_ROOT
$ export PMEM_MMAP_HINT=0x10000000000
```

The environment variable `PMEM_MMAP_HINT` sets a predefined virtual address for PM allocation. XFDetector depends on this functionality to identify PM accesses.

***PMDK Examples.*** We provide patches that create buggy PM programs and their inputs that trigger the bugs. The patches are under `xfdetector/patch` folder. Please execute the following commands to run our examples:

```
$ ./run.sh <WORKLOAD> <INITSIZE> <TESTSIZE> <PATCH>
```

- `WORKLOAD`: The workload to test.
- `INITSIZE`: The number of data insertions when initializing the PM image before testing starts.
- `TESTSIZE`: The number of data insertions when running the program with XFDetector.
- `PATCH`: The name of the patch that generates bugs for `WORKLOAD`. If empty, the script tests the original program.

The following example reports a cross-failure race bug (patch named as `btree_race1.patch`) in `btree`, where the program inserts 5 items during initialization and 5 more items during testing:

```
$ ./run.sh btree 5 5 race1
```

For a complete list of tests and corresponding input parameters, see `runallPMDK.sh`. You can also directly run the script to execute all available tests.

***Redis.*** Use script `runRedis.sh` under folder `xfdetector/` to run Redis:

```
$ ./runRedis.sh <TESTSIZE>
```

- `TESTSIZE`: The number of database insertions for testing.

***Memcached.*** Use script `runMemcached.sh` under folder `xfdetector/` to run Memcached:

```
$ ./runMemcached.sh <TESTSIZE>
```

- `TESTSIZE`: The number of database insertions for testing.

***Output.*** XFDetector reports all detected bugs after testing is complete. The output is dumped to the screen and the corresponding `<WORKLOAD>_<TESTSIZE>_debug.txt` file.

### A.7 Experiment Customization

XFDetector provides an interface for annotating programs and PM libraries. Please see Table 2 for the usage. When compiling the programs with XFDetector annotation, please add compiler flags following this example:

```
LIBS+= -L<XFDetector Root>/xfdetector/build/lib \
    -Wl,-rpath=<XFDetector Root>/xfdetector/build/lib \
    -lxfdetector_interface
CFLAGS+= -I<XFDetector Root>/xfdetector/include
```

## Acknowledgments

We thank our anonymous reviewers, Akhil Indurti, and Suyash Mahar for their valuable feedback. This work is supported by NFS and the SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP).

## References

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *TPDS*, 4(6):613–624, June 1993.

[2] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *ISCA*, 1991.

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

[4] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *SIGMOD*, 2017.

[5] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA*, 2014.

[6] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5):497–508, 2015.

[7] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *SOSP*, 2015.

[8] Shimin Chen and Qin Jin. Persistent B+-Trees in non-volatile main memory. In *VLDB*, 2015.

[9] Xianzhang Chen, Edwin H.-M. Sha, Ahmad Abdullah, Qingfeng Zhuge, Lin Wu, Chaoshu Yang, and Weiwen Jiang. UDORN: A design framework of persistent in-memory key-value database for NVM. In *NVMSA*, 2017.

[10] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *SOSP*, 2013.

[11] 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 *ASPLOS*, 2011.

[12] Nachshon Cohen, David T. Aksun, and James R. Larus. Object-oriented recovery for non-volatile memory. *Proc. ACM Program. Lang.*, 2(OOPSLA):153:1–153:22, October 2018.

[13] 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 *SOSP*, 2009.

[14] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *EuroSys*, 2014.

[15] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *FAST*, 2012.

[16] E. R. Giles, K. Doshi, and P. Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *MSST*, 2015.

[17] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous checkpointing of htm transactions in nvm. In *ISMM*, 2017.

[18] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *PLDI*, 2018.

[19] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *EuroSys*, 2017.

[20] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *ATC*, 2017.

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

[22] Intel. An introduction to pmemcheck. http://pmem.io/2015/07/17/pmemcheck-basic.html.

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

[24] Intel. PMDK man page: libpmem. http://pmem.io/pmdk/manpages/linux/v1.6/libpmem/libpmem.7.html.

[25] Intel. Redis. https://github.com/pmem/redis/tree/3.2-nvml, 2018.

[26] 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.

[27] Intel. Quick start guide: Configure Intel Optane™ DC persistent memory modules on Linux. https://software.intel.com/en-us/articles/quick-start-guide-configure-intel-optane-dc-persistent-memory-on-linux, 2019.

[28] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *ASPLOS*, 2016.

[29] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *HPCA*, 2017.

[30] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using NVM as virtual memory. In *IPDPS*, 2013.

[31] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *ASPLOS*, 2016.

[32] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *MICRO*, 2016.

[33] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High-performance metadata integrity protection in the WAFL copy-on-write file system. In *FAST*, 2017.

[34] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *SOSP*, 2017.

[35] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[36] Philip Lantz, Dulloor Subramanya Rao, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *ATC*, 2014.

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

[38] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *ASPLOS*, 2017.

[39] 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 *MICRO*, 2018.

[40] S. Liu, A. Kolli, J. Ren, and S. Khan. Crash consistency in encrypted non-volatile main memory systems. In *HPCA*, 2018.

[41] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. Janus: Optimizing memory and storage support for non-volatile memory systems. In *ISCA*, 2019.

[42] 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.

[43] David E. Lowell and Peter M. Chen. Free transactions with rio vista. In *SOSP*, 1997.

[44] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An RDMA-enabled distributed persistent memory file system. In *ATC*, 2017.

[45] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent Computing: Challenges and Opportunities. In *SNAPL*, 2017.

[46] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *PLDI*, 2015.

[47] 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 *PLDI*, 2005.

[48] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, 2014.

[49] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing legacy code to byte-addressable persistent memory. In *HotStorage*, 2017.

[50] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A framework to systematically test file-system crash consistency. In *HotStorage*, 2017.

[51] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1):94–162, 1992.

[52] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *OSDI*, 2018.

[53] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *ASPLOS*, 2017.

[54] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *LOPLAS*, 1(1):74–88, March 1992.

[55] Tri Nguyen and David Wentzlaff. PiCL: A software-transparent, persistent cache log for nonvolatile main memory. In *MICRO*, 2018.

[56] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *HotStorage*, 2018.

[57] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[58] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *PPoPP*, 2003.

[59] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *SOSP*, 2005.

[60] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *MICRO*, 2015.

[61] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *TOCS*, 15(4):391–411, November 1997.

[62] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *WBIA*, 2009.

[63] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *OSDI*, 2016.

[64] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*, 2014.

[65] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memeory. In *ASPLOS*, 2011.

[66] Benjamin Wester, David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Parallelizing data race detection. In *ASPLOS*, 2013.

[67] Michael Wu and Willy Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *ASPLOS*, 1994.

[68] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *SC*, 2011.

[69] 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 *ApSys*, 2016.

[70] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *ATC*, 2017.

[71] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, 2016.

[72] 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 *SOSP*, 2017.

[73] 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 *FAST*, 2015.

[74] M. Ye, C. Hughes, and A. Awad. Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories. In *MICRO*, 2018.

[75] 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 *MICRO*, 2013.