PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs

Sihang Liu
University of Virginia

Yizhou Wei
University of Virginia

Jishen Zhao
UC San Diego

Aasheesh Kolli
Penn State University
VMware Research

Samira Khan
University of Virginia

Abstract

Recent non-volatile memory technologies such as 3D XPoint and NVDIMMs have enabled persistent memory (PM) systems that can manipulate persistent data directly in memory. This advancement of memory technology has spurred the development of a new set of crash-consistent software (CCS) for PM - applications that can recover persistent data from memory in a consistent state in the event of a crash (e.g., power failure). CCS developed for persistent memory ranges from kernel modules to user-space libraries and custom applications. However, ensuring crash consistency in CCS is difficult and error-prone. Programmers typically employ low-level hardware primitives or transactional libraries to enforce ordering and durability guarantees that are required for ensuring crash consistency. Due to the reordering by the hardware, programmers cannot *test* whether the order specified in the CCS will *not* result in an ordering that *violates* the crash consistency requirement.

We believe that there is an urgent need for developing a testing framework that helps programmers identify crash consistency bugs in their CCS. We find that prior testing tools lack generality, i.e., they work only for one specific CCS or memory persistency model and/or introduce significant performance overhead. To overcome these drawbacks, we propose PMTest[1], a crash consistency testing framework that is both flexible and fast. PMTest provides flexibility by providing two basic assertion-like software checkers to test two fundamental characteristics of all CCS: the ordering and durability guarantee. These checkers can also serve as the building blocks of other application-specific, high-level checkers. PMTest enables fast testing by deducing the persist order without exhausting *all* possible orders. In the evaluation with eight programs, PMTest not only identified 45 synthetic crash consistency bugs, but also detected 3 new bugs in a file system (PMFS) and in applications developed using a transactional library (PMDK), while on average being 7.1× faster than the state-of-the-art tool.

CCS Concepts • **Hardware → Emerging technologies**; • **Software and its engineering → Software testing and debugging**.

Keywords Persistent Memory, Crash Consistency, Debugging, Testing

ACM Reference Format:
Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3297858.3304015

1 Introduction

Persistent Memory (PM) technologies offer the persistence of disks combined with performance close to that of DRAM, blurring the divide between memory and storage [34, 40, 45, 67]. PMs are expected to be placed alongside DRAM on the system's memory bus and be accessed via a byte-addressable load/store interface, providing an opportunity to manipulate persistent data directly in-place in memory. Programs can recover their updated in-memory persistent data even in the event of a crash (e.g., power failure). However, such a recovery requires a guarantee that persistent data is always in a consistent state – a requirement referred to as the crash consistency guarantee. A variety of applications have taken crash consistency into consideration. File systems carefully orchestrate meta-data management to ensure that the files are recoverable [10, 16, 42, 63, 66, 73], while databases use intricate logging mechanisms to provide ACID guarantees for transactions [1, 2, 23, 50, 65]. Apart from relying on file systems and databases for crash consistency [1, 2, 10, 16, 23, 42, 50, 63, 65, 66, 73], the advent of PMs makes it possible for applications to manage crash consistency directly using PM's load/store interface and thereby, improve performance by avoiding costly system calls. For this reason, a variety of custom crash-consistent applications [7, 17, 33, 70, 72] and user-space libraries (e.g., NV-Heaps [9], Mnemosyne [64], PMDK [33]) have been developed for PM systems. Moving forward, we expect that

[1] PMTest is available at https://pmtest.persistentmemory.org.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304015

PM systems will spur the development of many more custom crash-consistent kernel modules (like file systems), storage applications, and user-space libraries. Collectively, we refer to them as crash-consistent software (CCS) for PM systems.

However, programming in PM systems for crash consistency is hard and error-prone. The two fundamental guarantees required by any CCS are *durability* and *ordering*. A *durability guarantee* from the PM system is required to enforce data to reliably reach persistence. As the cache hierarchies are volatile in our current systems, simply executing a store instruction to a PM location does not ensure that the new value is persistent. To solve this problem, the x86 ISA introduced new optimized instructions (e.g., `clwb` [32]) to efficiently writeback cache lines to memory. We refer to the act of making a cache line persistent (through a writeback or other means [39, 52]) as a *persist operation*.

Enforcing ordering is another fundamental necessity for any CCS. An *ordering guarantee* from the PM system is required for CCS to explicitly order *persist operations* as the hardware can reorder instructions in the processor and cache hierarchy. For example, the commonly used undo logging mechanism [9, 38] requires the undo log entry to be created and persisted *before* the associated data gets modified. x86 systems provide ordering guarantees through the `sfence` instruction. However, different architectures provide durability and ordering guarantees through architecture-specific instructions [3, 32]. While developing a CCS for PM systems, programmers must carefully use these low-level primitives for correctness. Relying on such low-level, architecture-specific primitives to develop software is hard and error-prone. Even with the help of transactional libraries that build upon these low-level primitives [9, 33, 47, 64], programmers still need to understand the specification of the durability and ordering guarantees provided by these libraries to properly use them. The major difficulty arises from the fact that the order of persist operations executed in the hardware can be different from the program order. As a result, programmers cannot determine whether the crash consistency algorithm is correctly implemented, i.e., whether the order specified in the CCS will *not* result in a runtime ordering that *violates* the required ordering of the *persist* operations. We refer to the bugs that cause a CCS to fail recovery as *crash consistency bugs*.

We argue that CCS developers will greatly benefit from a testing infrastructure that can help identify the improper use of low-level primitives or high-level libraries. While prior works have developed tools to assist CCS development, they are all specific to certain file systems [43] or user-space libraries [55, 59]. These tools rely on exhaustive search space exploration of all possible ordering or binary instrumentation of the program, leading to a significant performance overhead. For example, Yat [43], a tool that tests Intel's persistent memory file system (PMFS [16]) can take more than 5 years to test all possible orderings in a trace with around 100k PM operations. In this work, we argue that an effective testing tool needs to meet two requirements. First, the testing mechanism needs to be fast so that programmers can reason about the durability and ordering of the persistent operations and detect bugs in the development phase. Second, the testing must support a myriad of CCS that will be built with various architecture-specific low-level primitives and high-level libraries. It also needs to support different persistency models that order persists in various ways. For example, Intel and ARM uses a strict ordering of writes [3, 32], while recent academic proposals relax this ordering [39, 52, 57]). In this work, we propose PMTest, a crash consistency testing framework that is, unlike prior work, both *flexible* and *fast*.

Flexible. Our key idea is based on the observation that regardless the difference in CCS (kernel modules, or custom applications using architecture-specific low-level primitives or high-level libraries), they all fundamentally rely on two types of operations in order to provide the durability and ordering guarantee: enforcing persisting a write and enforcing ordering between writes. To this end, we propose two low-level *checkers* that developers can debug their CCS with: `isPersist()` and `isOrderedBefore()`, that check whether *(i)* certain persistent objects have been persisted since their last update and *(ii)* if a certain *persist* operation has been ordered before another, enabling testing of the two fundamental properties of any CCS. Similar to assertions [20, 71] used in programs, these two checkers can be instrumented in the code, which provides a way to expose the ordering and durability of the persistent operations to the software (details in Section 4.4). On top of that, programmers can use the PMTest framework to build custom, high-level checkers in the software based on the two low-level checkers for different libraries and persistency models (details in Section 5). High-level checkers can automate the process of debugging CCS built with PM libraries.

Fast. PMTest enables high-speed testing by inferring the ordering of *persist* operations without exhaustively testing *all* possible orders. The key idea is to track the PM operations (e.g., writes, cache writeback, fence) at runtime and deduce the time interval during which a write may persist. An overlapping time interval for two write operations implies that the two writes are *not* strictly ordered; the ending time of the interval determines at what point in the program the write is guaranteed to persist.

We evaluate the capability of PMTest bug detection in two ways. First, PMTest detects 45 manually created bugs (synthetic and reproduced from the commit history) in WHISPER [52], a benchmark suite for PM. Second, PMTest detected 3 new bugs in a file system (PMFS) and in applications developed using a transactional library (PMDK). These bugs have been reported to Intel and have been fixed with proper credit to PMTest [30, 31]. Further, our experiments also reveal that PMTest checkers can help programmers understand the persistency guarantees of PM libraries.

Contributions. Our main contributions are as follows:

- We design and implement PMTest, a tool to detect crash consistency bugs in PM applications. To our knowledge, PMTest is the first tool that is both flexible and fast.
- PMTest is flexible as it enables the design of specific checkers in the software for different libraries and persistency models. Currently, PMTest supports user-space transaction memory libraries Mnemosyne [64] and PMDK [33] and Intel's kernel-space PM-optimized file system PMFS [16] under the x86 persistency model [32].
- PMTest is fast as it detects the violation in durability and ordering of PM operations without exhaustively testing all possible reorderings. Our evaluation shows that PMTest is 7.1× faster than the state-of-the-art tool [59].
- PMTest detects 45 synthetic/reproduced bugs and found 3 new bugs in PMDK applications [33] and PMFS [16].

2 Motivation

In this section, we first discuss the difficulties in programming crash-consistent software for persistent memory systems and then introduce the requirements for testing these programs.

2.1 Difficulties in Programming CCS

There are two fundamental guarantees required by any crash-consistent software (CCS). *(i)* A *durability guarantee* to make data reliably persistent, and *(ii)* an *ordering guarantee* to explicitly enforce the ordering of writes. For example, the commonly used undo logging mechanism [9, 38] requires its log entry to be *durable before* the in-place update. As the cache hierarchy of processors is volatile, hardware vendors provide PM-specific instructions to writeback data from the cache to memory to ensure *durability*. For example, Intel extends the x86 ISA with new instructions (e.g. `clwb`) to enforce persistent data writeback to PM [32]. Similarly, ARM implements the `DC CVAP` instruction [3] that writes back data to the persistence. We refer to the act of making a cache line persistent (through a writeback or other means [39, 52]) as a *persist*. Similarly, as the processor can reorder instructions in the pipeline and memory hierarchy, Intel provides ordering guarantees through the `sfence` instruction [32] which ensures a strict ordering between persists before and after the fence. Therefore, the combination of a `clwb` and an `sfence` issued after a write to a cache line ensures that the new value of the cache line has persisted before any subsequent instructions. In the rest of this paper, we will refer to the combination of "`clwb; sfence;`" as a `persist_barrier` for simplicity. Apart from industry implementations, there have been proposals from the academia that target better performance with relaxed ordering and durability guarantee. For example, a recent work, hands-off persistence system (HOPS) [52], proposed new relaxed fences to decouple the ordering guarantee (provided by `ofence`) from the durability guarantee (provided by `dfence`).

```
1 void ArrayUpdate(int index, item_t new_val) {
2   backup.val = array[index]; //Backup the old value
3   backup.valid = true;       //Set the backup as valid
4   persist_barrier();         Missing persist_barrier()
5   array[index] = new_val;    //Update to the new value
6   backup.valid = false;      //Set the backup as invalid
7   persist_barrier();
8 }
```

(a)

```
1 void appendList(item_t new_val) {
2   TX_BEGIN {
3     node_t *new_node = makeNode(new_val);//Create a new node
4     TX_ADD(list.head, sizeof(node_t*));  //Backup old head in log
5     List.head = new_node;                //Update head
6     List.length++;   Missing backup:     //Increment length of list
7   } TX_END     TX_ADD(&list.length,sizeof(int));
8 }
```

(b)

Figure 1. Buggy examples using (a) low-level functions and (b) a transactional interface.

Programming with Low-level Primitives. With the support from these low-level primitives, programmers can ensure crash consistency by enforcing a specific order of persists. Unfortunately, it is hard to implement the intended ordering using these low-level primitives even when the programmers understand the semantic of the crash-consistency support. We provide a simple example to show the difficulty associated with using these low-level primitives. Figure 1a shows a function that tries to update the value of an array element in a crash-consistent manner. The program takes the undo logging approach that backs up the data before performing the modification in-place, such that there is always a consistent copy (either the backup or the original data) for recovery. Following this approach, it first creates a backup copy (line 2) and sets it to be valid (line 3). Then, it persists the backup (line 4), followed by updating the array index in place (line 5), and invalidates the backup copy (line 6). Finally, it persists the in-place update and invalidation (line 7). This example seems correct as it places a `persist_barrier` after the backup and after the in-place update assuming that these barriers will ensure that the update is only performed after the backup gets persisted. However, it still misses two `persist_barriers`: *(i)* one right after the creation of the backup copy (between line 2 and 3), and *(ii)* another right after updating the new array index (between line 5 and 6). Omitting *any one* can render the array unrecoverable in event of a failure. If a failure occurs at line 6, it is possible that due to hardware reordering `valid` has persisted while the actual data has not. Therefore, after recovery, the array will treat the stale value in memory as the updated one. As the example shows, using such low-level primitives is hard, especially for complex code bases. There is a need for a testing framework to identify and resolve such bugs. Next, we will show that using high-level crash consistency mechanisms like transactions are no panacea.

Programming with High-level Interface. To abstract away the low-level implementations and improve programmability, prior works have provided libraries for CCS [9, 33, 64]. For example, with the transactional interface from PMDK [33], programmers can create a failure-atomic transaction with a

pair of TX_BEGIN and TX_END, and use TX_ADD() to create a backup (snapshot) of the persistent object before modifying it such that the object can roll back to its old data value if the transaction fails to complete due to a crash. The example in Figure 1b shows an insertion function of a linked list using a transactional interface that appends a new node to the head. The code seems correct as the programmer wraps up the entire procedure into a transaction and adds the head to the log for recovery. However, this function is *not* crash consistent as the programmer mistakenly assumes that the length of the linked list will get persisted automatically and misses backing it up (via a TX_ADD()). Therefore, in event of a failure, the transaction will not be able to recover the correct length of the list. The correct implementation should call TX_ADD() to backup the length field before line 6. We argue that even though transactional libraries are supposed to make persistent programming easier, it is still very likely to introduce subtle crash consistency bugs.

In the first example (Figure 1a), the programmer is intended to set/unset the valid bit *after* persisting the backup/update, but misses the persist_barriers. Similarly, in the second example (Figure 1b), the programmer intends to make both the linked list and its length recoverable, but forgets to backup the length. We conclude that the major difficulty in detecting crash consistency bugs in CCS is that it is difficult to ensure the program operates on its persistent data in the way that programmers intend to. Even if the algorithm for crash consistency is correct, the implementation can be wrong as the programmers cannot directly infer how writes to PM get persisted from looking at the code. Fences and writeback operations do not provide an intuitive interface for programmers to reason about *(i)* whether a memory location/object has persisted, and *(ii)* the order in which different memory locations/objects persist, the two fundamental requirements to reason about crash consistency.

2.2 Requirements for CCS Testing Tools

We believe that programmers will greatly benefit from a testing framework to help identify crash consistency bugs. Such frameworks should ideally meet the following requirements.

Flexible. We expect that PM systems will spur the development of many custom CCS and a testing framework must be flexible to support as many as possible. First, there are three types of CCS: *(i)* user-space applications using high-level libraries such as NV-Heaps [9], Mnemosyne [64], and PMDK [33], *(ii)* user-space applications using ISA-specific low-level primitives, such as PM database [2] and key-value stores [50], and *(iii)* kernel-space file systems using low-level primitives, such as PMFS [16] and NOVA [68]. Second, the other variation in CCS comes from the different ordering and durability guarantees provided by different PM systems, or more specifically, different persistency models that define the rules for the order of persists [57] (e.g., the strict persistency model from x86 [32] and the relaxed model proposed

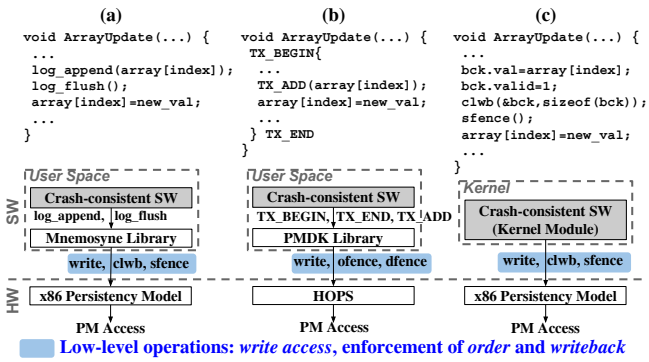


Figure 2. Different PM system stacks and sample codes.

by HOPS [52]). The persistency model is enforced using low-level primitives from the underlying hardware, e.g., clwb and sfence in x86, and ofence and dfence in HOPS. In the future, we expect to see a great variety of CCS running on various PM systems. Figure 2 shows three possible system stacks and their code examples: (a) a CCS developed on top of the Mnemosyne library [64] runs on a system with x86 persistency model, (b) a CCS built with the PMDK library [33] runs on the HOPS persistency model that supports more relaxed fences [52], and (c) a persistent kernel module using low-level functions (e.g., PMFS [16]). Ideally, a testing framework should be flexible enough to support all kinds of CCS running on a variety of PM systems.

Fast. We identify that an efficient crash consistency testing mechanism needs to meet two performance requirements. First, a crash-consistency testing solution for CCS needs to be able to identify issues in the programs as fast as possible. Second, an efficient crash-consistency testing mechanism needs to maintain a low performance overhead to the target program; it is favorable that programmers can reason about their code at runtime and modify the code as necessary to reduce the overhead of post-production patching [53]. However, no prior tools can meet both the flexibility and fast requirements.

We categorize the prior tools into three groups. First, there is a large body of crash consistency bug detection tools developed for conventional file systems running on block devices [6, 18, 19, 41, 51, 61]. Unfortunately, these tools are designed for block-addressable file systems [6, 18, 19, 41, 51, 61], and therefore, cannot be applied to PM-specific CCS. Second, the tool, Yat [43], that tests Intel's PM-based file system PMFS [16] executes at an extremely slow speed because it takes an exhaustive method in bug detection. It permutes all possible persist reorderings to detect if a particular ordering can recover consistently after a crash. Such an exhaustive method is extremely slow and according to the authors, can take more than five years to test an application with around 100k PM operations [43]. Third, there have been faster testing tools developed for specific PM libraries. For example, Pmemcheck [59] (around 20x slowdown) and Persistence Inspector [55] are binary instrumentation platforms

designed specifically for the PMDK library. They provide built-in checkers for PMDK operations and cannot be easily extended for other user-space libraries or kernel-space system software. Table 1 summarizes the capabilities of these tools and it is evident that they cannot satisfy both requirements of speed and flexibility.

Tool Name	Speed	Flexibility	Target Software	Kernel?
Yat [43]	Low	Low	PMFS [16]	Yes
Pmemcheck [59]	Medium	Low	PMDK [33]	No
PMTest (this work)	High	High	Any CCS	Yes

Table 1. Tools for testing CCS.

3 Key Ideas of PMTest

In this work, we propose PMTest, a framework for detecting crash consistency bugs in different CCS running on a variety of PM systems. First, we present our high-level ideas in testing CCS. Then, we discuss how these key ideas are applied to PMTest.

3.1 Key Ideas in Testing Crash Consistency

The *goal* of this work is to design a crash consistency testing framework that is, unlike prior works, both *flexible* and *fast*. Our keys ideas to meet these requirements are:

Flexible. We observe that regardless of the difference in the CCS (kernel module, user-space library, or custom application using architecture-specific low-level primitives), they all fundamentally rely on two types of operations in order to provide the durability and ordering guarantee: enforcing a memory location persists and enforcing ordering between persists. Figure 2 shows that at the lowest level, all three CCS rely on low-level primitives that provide these two guarantees (shown by the blue arrows). Our key idea is to provide two generic "checkers" that programmers can instrument their code with to verify whether certain memory locations/objects have been persisted since the last write to them and the order in which certain memory locations/objects have persisted. These generic checkers allow programmers to ascertain the state of the PM on any kind of PM system, making it easy to reason about crash consistency. The two generic checkers are: *(i)* `isPersistent()` checks whether certain memory locations/objects have been persisted since their last update; *(ii)* `isOrderedBefore()` checks whether a certain address has been persisted before another (details in Section 4.4).

Similar to the commonly used assertions [20, 71], these two checkers can be placed in the code, providing a way to expose the ordering and durability of the PM operations at the application level. Figure 3a and 3b demonstrate how these two checkers make the ordering information visible to applications in systems using the x86 and HOPS persistency model, respectively. Even though the systems are different, the same two basic, low-level checkers in both examples checks: *(i)* whether A persists before B, and *(ii)* whether both A and B have been persisted at the end. PMTest, under-the-hood uses PM system-specific information to determine if the checker conditions have been met on the system under test.

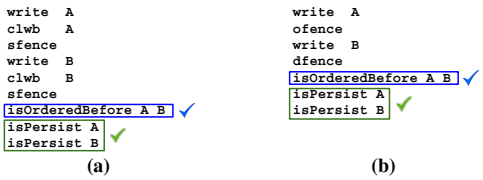


Figure 3. Checking mechanism based on the semantics of (a) the x86 persistency model [32] and (b) HOPS [52].

Fast. Our key idea is to track the PM operations (e.g., write, `clwb`, `sfence` in x86 systems) at runtime and deduce the time interval during which a write may persist. We refer to this time interval as a *persist interval*. PMTest's superior performance comes from validating the programmer specified checkers from the inferred *persist interval*, rather than checking all possible orderings of relevant persists. The rules that deduce the persist interval and validate the checking of durability and ordering guarantee for a certain persistency model are referred to as *checking rules*. For example, in x86 systems, a PM write may persist any time between its execution and a subsequent `sfence`, assuming that there exists an intervening `clwb` to the associated cache line in between the write and `sfence`. This is due to the fact that the hardware can reorder operations as long as they are executed before the `sfence`. Note that even though the hardware can re-order instructions, x86 implicitly guarantees the ordering of a write operation and a subsequent `clwb` to the same address [32]. Therefore, the persist interval of a write can span from the last `sfence` to the subsequent `sfence` that comes after the associated `clwb`. To validate checkers, we use the persist intervals for the relevant memory locations to infer if the checker conditions are being met. We break a thread's execution into epochs separated by an `sfence`. We use an epoch as a unit of time and have a timestamp increment at every `sfence`. A persist interval of (E_1, E_2) suggests the corresponding write may persist any time between epoch number E_1 and E_2. Therefore, the checking rule for `isPersist()` is defined as determining if the persist interval of the associated memory location ends *before* the checker. Similarly, the `isOrderedBefore()` is checked by determining if one persist interval ends *before* the other starts.

We provide an example to show how to infer the *persist interval* from the trace and how it can be used by our two basic checkers in an x86 system. Figure 4a shows a trace of PM operations, where the programmers want to check two issues: if A always persists before B, and if B has been persisted after the last `sfence`. Assuming the first `sfence` starts the first epoch ($E = 1$), the persist interval for address A is $(1, 2)$, as the write to address A, and the subsequent `clwb` are both issued before the next `sfence` (the start of the second

epoch, $E = 2$). For address B, the persist interval is $(1, \infty)$ as the write to B is in the first epoch, so it may persist as early as the first epoch. However, without a subsequent `clwb` for address B, it is never guaranteed to persist (at least in the code snippet). As the persist intervals of A and B overlap, the checker, `isOrderedBefore()` for A persisting before B fails. The subsequent `isPersist()` for address B also fails as the persist interval for B extends to ∞,

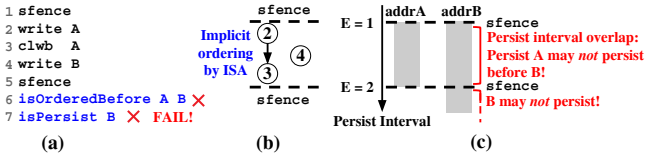


Figure 4. (a) A trace of PM operations. (b) The order between PM operations. (c) The persist interval of writes.

3.2 Integrating the Key Ideas into PMTest

So far, we have introduced the key ideas that ensure both flexibility and high-speed testing. Next, we introduce how we apply our key ideas to the two major steps of PMTest:

Program Annotation. The assertion-like, low-level checkers: `isOrderedBefore()` and `isPersist()`, provide a system-independent interface for testing. Figure 5a shows how to place these checkers to detect crash consistency bugs. Similar to using low-level primitives for programming CCS, using these low-level checkers requires manual effort. Therefore, to ease programmers' burden, PMTest provides high-level checkers that are built on top of the low-level ones. Figure 5b shows a pair of high-level checkers placed before and after a transaction, which *automatically* detects whether all modified persistent objects have been written back at the end of a transaction. Programmers (e.g., PM library developers) can also build their custom checkers using our low-level checkers (details in Section 5.1). We show that these high-level checkers can effectively detect bugs with minimal programmer's effort in Section 6.3.

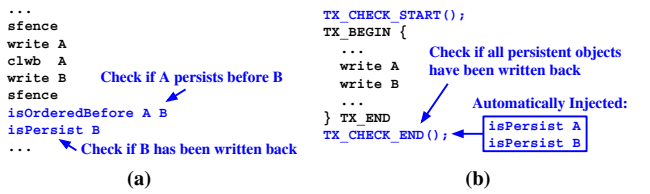


Figure 5. Examples of testing programs using (a) the fundamental checkers and (b) checkers for transactions.

Runtime Testing. PMTest determines whether the injected checkers are met or not by inferring the interval in which a write to PM can become persistent based on the underlying persistency model. The superior performance makes it possible to perform testing during execution time. For better efficiency, PMTest pipelines the execution of CCS (the test program) and the checking engine by running them on different threads. The test program under execution produces a trace of all the key events. Meanwhile, the checking engine lags behind program execution and consumes the trace produced (details in Section 4.4). Decoupling program execution from checker validation provides a marked improvement in performance.

4 Implementation of PMTest

This section describes the implementation of PMTest and how it can be integrated into a real system to perform testing.

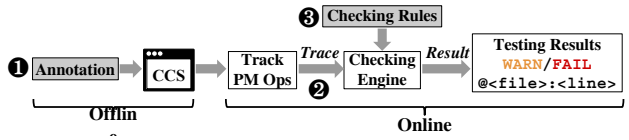


Figure 6. A high-level view of PMTest (shaded components can be customized by programmers).

4.1 Overview of PMTest

Figure 6 illustrates a high-level view of PMTest. The procedure of testing a program consists of *offline* and *online* steps. In the *offline* step, programmers annotate the CCS using low-level and/or high-level checkers following the program specification of the crash consistency mechanism (step ❶). For example, low-level checkers should be inserted to check the programmer intended crash-consistent behavior, where the high-level checkers for transactions can be added by wrapping up the transactions (as shown in Figure 4). In the *online* step, PMTest executes with the annotated (and compiled) CCS. During execution time, PMTest tracks PM operations in the application and passes the trace to the checking engine (step ❷, details in Section 4.3). The checking engine tests whether the trace meets the requirements specified by the checkers (step ❸, details in Section 4.4). The checking engine depends on the checking rules to detect the bugs. We discuss the rules for x86 systems in Section 4.4 (already integrated in PMTest) and the rules for HOPS [52] in Section 5.2. The new checking rules for other persistency models can be integrated into PMTest by programmers. The checking engine reports `WARNING` outputs for performance bugs (e.g., redundant writebacks) and `FAIL` outputs for crash consistency bugs (e.g., missing a fence), together with the file names and line numbers of the failing checkers.

4.2 PMTest Interface

PMTest incorporates a flexible software interface that is C and C++ compatible. Table 2 summarizes the functions offered by PMTest. There are four types of functions. The first category is for initializing and enabling the testing functionalities of the framework. Programmers can select the region for testing by wrapping the code with a pair of `PMTest_START` and `PMTest_END` functions. The second

	Function Name	Description
Framework	PMTest_INIT	Initialize PMTest
	PMTest_EXIT	Exit and clean up PMTest
	PMTest_THREAD_INIT	Initialize per thread PMTest tracking
	PMTest_START	Enable PMTest tracking and testing
	PMTest_END	Disable PMTest tracking and testing
PM Object	PMTest_EXCLUDE	Remove a persistent object from testing scope
	PMTest_INCLUDE	Add a persistent object back to testing scope
	PMTest_REG_VAR	Register the address and size of a variable name
	PMTest_UNREG_VAR	Unregister a variable name
	PMTest_GET_VAR	Get the address and size of a variable by its name
Comm.	PMTest_SEND_TRACE	Send the current trace to PMTest checking engine and start a new trace
	PMTest_GET_RESULT	Block the program execution until all existing traces have been tested
Checker	isPersist	Check if a persistent object has been persisted
	isOrderedBefore	Check the order of two persists
	TX_CHECKER_START	Start checking transactions
	TX_CHECKER_END	End checking transactions

Table 2. Summary of PMTest functions.

category of functions allows programmers to operate on persistent objects. By default, all accesses to PM between `PMTest_START` and `PMTest_END` are tracked by PMTest. Programmers may exclude objects from tracking using `PMTest_EXCLUDE() function`. Already excluded objects can be tracked again using `PMTest_INCLUDE()`. To allow programmers check the persistency status of a variable outside its scope (e.g., outside the function where it is declared), we provide three functions: `PMTest_REG_VAR`, `PMTest_UNREG_VAR`, and `PMTest_GET_VAR` that allow programmers to register the address of a persistent object with a name and check its persistency status later. The third category of functions enables the communication from the test program to the checking engine. Programmers can divide a program into independent sections (e.g., transactions) using `PMTest_SEND_TRACE` for better testing speed. Once the execution of a section is complete, PMTest can start testing it on a separate thread while the program is executing the next section. The function `PMTest_GET_RESULT` blocks the program execution until all previously generated traces have been tested. The last category of functions are checkers, including two low-level checkers: `isOrderedBefore()` and `isPersist()`, and the high-level checkers for transactions. The high-level checkers for PMDK test three issues: *(i)* if a transaction has completed, *(ii)* if the persistent objects within the transaction have been added to the undo log before modification, and *(iii)* if there are unnecessary writebacks and redundant logs that constitute the performance bugs.

4.3 Operation Tracking

A trace in PMTest consists of the PM operations executed by CCS and the checkers placed by programmers. Each PM operation in the trace has associated metadata that consists of the operation type, memory address, operation size and the file and line number of this operation. Similarly, the metadata for each checker consists of the type of checker, the address and size of the persistent object that the checker is testing in PMTest. All PM operations and checkers are recorded in the trace in program order. When the program calls `PMTest_SEND_TRACE()`, PMTest passes the current trace to the backend checking engine and starts a new trace. In our evaluation with testing the PM benchmark suite, WHISPER [52], we extend the tracking mechanism of WHISPER by adding PMTest tracking functions that generate the aforementioned metadata for PM operations (e.g., writes, `clwb` and `sfence` in x86) to the WHISPER's PM operation macros. For other CCS, it is possible to either integrate a WHISPER-like tracking mechanism or use a toolchain (e.g., through an LLVM [44] pass) that injects a tracking function for each PM operation.

4.4 The Checking Engine

After generating a trace of PM operations and checkers from the application, the next step is to validate the trace against the specified checkers. At the high-level, the checking engine tracks a *persistency status* for each persistent object in the trace. During testing, PMTest sequentially iterates over the trace. If the trace component is a PM operation, PMTest updates the persistency status; if the trace component is a checker, PMTest examines the persistency status to determine whether the asserted condition is met or not. Next, we describe the details of maintaining the persistency status in PMTest, and discuss how it updates and checks the status in an x86 system.

Persistency Status. PMTest maintains a *shadow memory* that represents the persistency status of each *modified* address. As PMTest traces and checks PM operations at a coarse granularity, it maintains the shadow memory as an interval tree [13], where the address is the interval and persistency status is the value in the interval, As a result, update and lookup operations to the shadow memory have a complexity of O(log n), where n is the length of the trace. As traces are independent, every trace has its shadow memory. To track the persistency status, the shadow memory keeps two types of structures, a *global status* for the entire system, and a *local status* for each address in the shadow memory. The following is the description of the fields for x86 systems:

- **`global_timestamp` (global status):** A global epoch counter that is incremented on every `sfence` encountered in the trace.
- **`persist_interval` (local status):** The interval in which certain memory location(s) may persist.
- **`flush_interval` (local status):** The interval in which certain memory location(s) may be explicitly written back to PM.

Update to Persistency Status. PMTest iterates over the trace and performs the following updates to the persistency status for each PM operation:

- **`write(addr,size)`** modifies an address range of $[addr, \texttt{addr} + size)$ in the shadow memory. It first clears all

existing `persist_intervals` and `flush_intervals` within the address range and sets the `persist_intervals` as $(global\_timestamp, \infty)$. That is, this write may persist at any time moving forward.

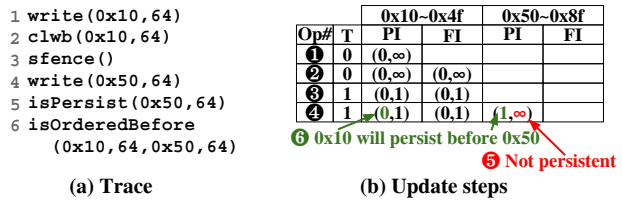
• **`clwb(addr,size)`** writes back an address range of $[addr, addr+size)$ and the `flush_interval` is set as $(global\_timestamp, \infty)$. That is, a writeback for these addresses has been issued and it may happen at any time moving forward. If there is an existing `flush_interval`, PMTest raises a `WARNING` (Section 5.1.2).

• **`sfence`** enforces the ordering of prior `write` and `clwb` operations. First, it increments the `global_timestamp`. Second, it updates the `flush_interval` of prior `clwb`s so that the intervals end at the current `global_timestamp`, i.e, the writeback is complete. Third, it updates the `persist_interval` of prior `clwb`s so that the intervals end at the current `global_timestamp`, i.e, the write persisted.

Checking Rules. Similarly, when encountered a checker in the trace, PMTest applies the following checking rules:

• **`isPersist(addr,size)`** checks whether data in the address range $[addr, addr+size)$ has been written to PM by checking whether the `persist_intervals` in this address range end before the current `global_timestamp`.

• **`isOrderedBefore(addrA,sizeA,addrB,sizeB)`** checks whether *all* `writes` to $[addrA, addrA+sizeA)$ can persist before any `write` to $[addrB, addrB+sizeB)$ by checking if any of the `persist_intervals` in $[addrB, addrB+sizeB)$ overlap with any of the those in $[addrA, addrA+sizeA)$.



(a) Trace (b) Update steps

Figure 7. An example of checking a trace.

Example. Figure 7a shows a sample trace, and Figure 7b shows how each operation (OP#) updates the PMTest persistency status, including `global_timestamp` (T), `persist_intervals` (PIs), and `flush_intervals` (FIs). Initially, T is 0.

Line 1: The write updates the PI for address 0x10 to $(0, \infty)$.
Line 2: The `clwb` updates the FI for address 0x10 to $(0, \infty)$.
Line 3: The `sfence` first increments the timestamp T. Then, it updates the FI of its preceding `clwb` to $(0, 1)$, indicating this writeback will take effect before line 3. It also updates the PI for 0x10 to $(0, 1)$, indicating that this write has persisted.
Line 4: The `write` updates the PI for address 0x50 to $(1, \infty)$.
Line 5: The `isPersist()` checker examines the PI of 0x50. As $(0, \infty)$ does not end before the current T, this checker reports a `FAIL` output as indicated by the red arrow.
Line 6: The `isOrderedBefore()` checker compares the PIs of 0x10 and 0x50. As they do not overlap, this checker passes as indicated by the green arrow.

Execution of The Checking Engine. To reduce the overhead in the runtime testing, PMTest adopts a multithreaded checking mechanism consists of a *master thread* and a pool of *worker threads*, as shown in Figure 8a. The master thread dispatches the traces passed from the program under testing (details about communication between the program and PMTest in Section 4.5) to the task queue of the worker threads following a round-robin scheduling algorithm. Each worker thread tests its trace independently and sends the testing result back to the result queue in the master thread. Figure 8b demonstrates the workflow of this mechanism. The program first creates and initializes an instance of PMTest by calling `PMTest_INIT()` (step ❶). Then, the program starts the execution of transaction 1 (step ❷). After transaction 1 (TX1) completes, the program passes its trace to PMTest by calling `PMTest_SEND_TRACE()` (step ❸). Then, PMTest immediately dispatches this trace to a worker (worker 1) thread in the worker pool. The worker thread tests the trace and completes (step ❺). In the meanwhile, PMTest receives and tests the trace of TX2 using worker 2 (step ❻).

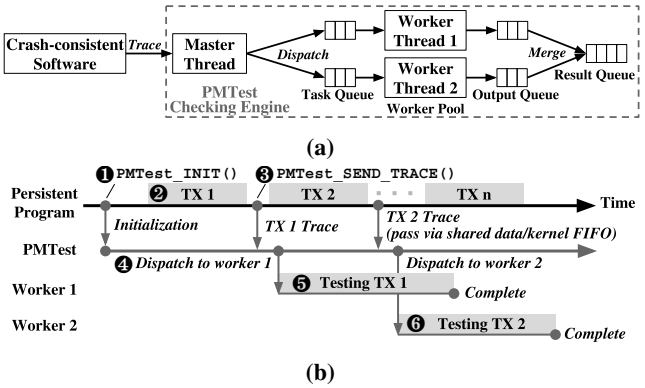


Figure 8. (a) The master and worker threads and (b) the workflow of PMTest.

4.5 System Integration

In this section, we describe PMTest's mechanism for user-space programs and kernel modules.

User-Space CCS. Figure 9a shows the system stack of testing a user-space CCS. The user-space CCS runs in the same process as the PMTest checking engine. To efficiently pass traces from CCS to the checking engine, we use a thread-safe, concurrent queue, where CCS pushes the traces to the queue and the testing module pops the head of the queue. PMTest also supports multithreaded programs. To manage the tracking of traces on different threads, PMTest maintains a per-thread data structure that maintains the trace of different threads. To initialize this structure, the programmers need to call `PMTest_THREAD_INIT()` when a thread is created. Note that PMTest only detects crash consistency bugs that is due

to incorrect PM operations in one thread. We leave the crash consistency issues due to improper thread synchronization as a future work.

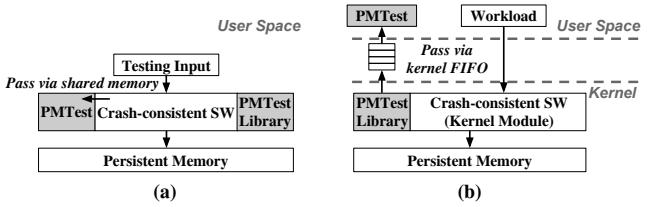


Figure 9. System integration of PMTest for (a) user-space programs and (b) kernel modules.

Kernel Modules. Crash-consistent kernel modules typically manage persistent data for user-space applications running on top (e.g., serve as a file system). Figure 9b illustrates how PMTest is integrated to test kernel modules. During execution, PMTest performs tracking in the kernel module in the same way as user-space programs. However, kernel programming has limited library support and has a strict constraint on the runtime performance. Therefore, PMTest checks the traces in the user space. To efficiently pass the trace from the kernel to the user-space checking engine, we use a kernel FIFO [12, 62] (created as `/proc/PMTest`) with 1024 trace entries. Currently, PMTest only tracks PM operations in one thread of the kernel module due to the limitation of kernel thread libraries. To prevent an exceptional case where the kernel FIFO becomes full and rejects new traces, PMTest maintains an interruptible wait queue [12] in the library. The kernel module put itself on the wait queue if the kernel FIFO is full. It gets interrupted and resumes execution when the FIFO is less than half full.

5 Flexibility of PMTest

So far, we have discussed the design of PMTest that enables fast testing for both user-space CCS and kernel modules. In this section, we discuss how PMTest further enables testing of different libraries and systems.

5.1 Implementation of Customized Checkers.

Customizing checkers can ease programmers' burden on debugging and improving the capability of PMTest. To implement more checkers, programmers need to add new methods to the checking engine module, which can be built on top of the existing low-level checkers. If the customized checker requires tracking more operations than the ones have been tracked by PMTest, the programmer can extend our tracking interface. We first present our high-level checkers designed for PMDK [33], and then present other checkers that detects performance bugs.

5.1.1 Library-Specific Checkers.

Library-specific, high-level checkers can automate the debugging for CCS developed with high-level libraries. We implement the following checkers for PMDK transactions. While these two checkers are designed for the PMDK transactions, they can be easily extended to other transactional libraries.

Check Incomplete Transactions. A typical bug in using transactions is the program fails to persist all updates when the transaction ends. To detect this type of bugs, we provide a pair of functions `TX_CHECKER_START` and `TX_CHECKER_END` that let programmers label the scope of the transaction. The `TX_CHECKER_END` automatically injects `isPersist()` for all modified persistent objects at the end of the trace for this scope. Using this checker, programmers can make sure that all transaction updates have persisted. Programmers can *exclude* the updates that do *not* require crash consistency protection in the transaction using the `PMTest_EXCLUDE()` function.

Check Missing Backup Logs. Another typical bug in using transactions is that programmers forget to log persistent objects before they get modified (e.g., the bug in Figure 1b). The correct implementation should use `TX_ADD()` to log persistent objects before modifying them, such that these objects can be recovered in event of a failure and be written back when the transaction ends. To detect such bugs, we extend the PMTest library to track the objects logged by `TX_ADD()` (or functions with similar functionality), together with other operations. The checking engine maintains another interval tree, *log tree*, that stores and tracks the logged memory addresses. When testing a trace from a transaction, the checking engine examines if the addresses under modification exist in the *log tree* before they get modified by a `write`.

5.1.2 Performance Checkers.

We provide the implementation of two checkers that detects unnecessary operations that can cause performance slowdown. PMTest reports a warning (`WARN`) when detecting such performance bugs.

Check Unnecessary Writeback. Enforcing the writeback of unmodified data can cause performance degradation. A typical scenario is coarse-grain writeback of persistent objects. Another possible scenario is that programmers writeback the same persistent object twice. The checking engine detects this types of bugs automatically when testing traces. The first case can be detected if a `clwb` operates on a memory location that does not yet have a `persist_interval`, i.e., writing back a PM location that has not been modified. The second case can be detected if a `clwb` operates on a memory location with an existing `flush_interval`, i.e., placing a second `clwb` after an existing one to the same PM location.

Check Duplicated Log. Logging the same persistent object more than once is unnecessary and can cause performance degradation. We implement a checker to detect this performance bug for PMDK transactions. When the program logs a persistent object, PMTest looks up the address of this object in the *log tree*. If it already exists, PMTest reports a `WARNING`.

5.2 Adaption to Other Persistency Models.

To adapt PMTest to other persistency modules, programmers need to track new system-specific PM operations and add new checking rules for these operations. Implementation new checking rules may require changing the global and local status fields in the shadow memory.

Recent works have proposed alternative persistency models that feature better performance and flexibility [37, 39, 52]. The hands-off persistence system (HOPS) [52] introduces two new primitives: `ofence` and `dfence`. The light-weight `ofence` guarantees all preceding write accesses reach PM prior to all write accesses after it; the heavier `dfence` stalls the processing until all writes to PM have been persisted. As PMTest provides a generic API for checkers, we only need to change the fields in the shadow memory and implement new rules in the backend checking engine. In the shadow memory, we still keep the `global_timestamp` and the `persist_interval`, but remove the `flush_interval` as HOPS does not use `clwb` and `sfence` to enforce ordering and durability. Then, we make the following *updates* to the rules in Section 4.4:

- **`ofence`** ensures the persist order without writing back the data from cache to PM. Therefore, this operation increments the `global_timestamp`.
- **`dfence`** ensures both ordering and writeback. It first increments the `global_timestamp`, and then updates the `persist_intervals` of prior writes to end at the current `global_timestamp`.
- **`isPersist(addr,size)`** checks if a `write` has persisted by checking whether the `persist_intervals` in address range $[addr, addr+size)$ end before the current `global_timestamp`.
- **`isOrderedBefore(addrA,sizeA,addrB,sizeB)`** checks whether the `write` to `addrA` persists before the one to `addrB`. As the fences already ensure persist order, PMTest checks whether all the `persist_intervals` in range $[addrA, addrA+sizeA)$ start before those in $[addrB, addrB+sizeB)$.

6 Evaluation

In this section, we evaluate the performance and bug detection capability of PMTest on a real system.

6.1 Methodology

To evaluate the performance and bug detection of PMTest, we use a real system as shown in Table 3. We use a set of battery-backed NVDIMMs as the PM and map them to the system following the method in [58]. We use CCS from the WHISPER benchmark suite [52] to evaluate both performance and bug detection. PMTest performs testing using one worker thread unless explicitly indicated. The execution times shown in this section are the average of ten runs.

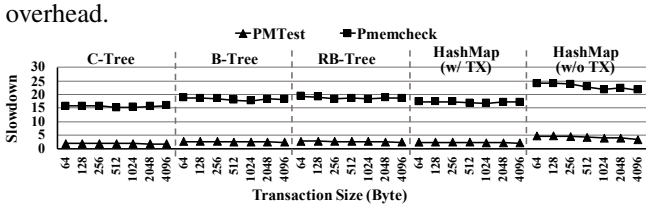
Server	HP ProLiant DL360 Gen10
Processor	Intel Skylake, 2.1GHz, 8 cores, 16 threads, 11MB L3 [28]
Memory	Volatile: 64GB DDR4, 2666MHz Non-Volatile: 64GB Battery-backed NVDIMM
OS	Ubuntu 14.04, Linux kernel 4.4.135
Compiler	gcc/g++ 4.8.4, O3 optimization

Table 3. System Configuration.

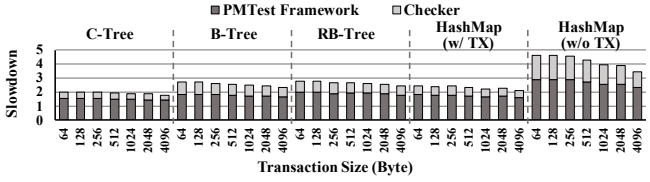
6.2 Performance Evaluation

6.2.1 Microbenchmark

We evaluate PMTest using five PMDK-based single-threaded microbenchmarks. We test each program with 100K insertions (each insertion is a transaction). Figure 10a compares PMTest with Pmemcheck. It is important to note the checkers used for PMTest provides *higher* bug-detection capabilities than those present in PMDK. The x-axis varies the size of the transaction and the y-axis shows the execution time normalized with the original versions without any testing tool. First, PMTest is 5.2-8.9× faster than Pmemcheck (7.1× avg.). Second, as the transaction size increases, the overhead in PMTest decreases as it tracks PM operations at a coarse granularity. In comparison, the slowdown from Pmemcheck does not change noticeably as it is based on the low-level binary instrumentation. Third, the overhead from the non-transactional HashMap is higher than other cases due to its more intensive use of low-level PM operations. We further present the overhead breakdown of PMTest as a stack diagram in Figure 10b, where the bottom bar shows the basic overhead from tracking PM operations and running the PMTest framework, and the top bar shows the extra overhead from the checkers. As PMTest decouples the checking from program's execution, checking only contributes 18.9%-37.8% of the total overhead. We conclude that PMTest has a relatively low performance overhead.



(a) Performance of PMTest vs. Pmemcheck.



(b) Overhead breakdown of PMTest.

Figure 10. Performance of testing microbenches.

6.2.2 Real Workloads.

We evaluate three real workloads shown in Table 4, where each of them has its own load-generating client(s). We place

the checkers to test whether all updates in the transactions (as specified by WHISPER) are persistent in PMFS [16] and Mnemosyne [64], and use our transaction checkers in Redis. Figure 11 shows the performance of these workloads running with PMTest. The y-axis shows the execution time normalized to the original versions without any testing tool. The slowdown from PMTest is between 1.33-1.98× (1.69× avg.). As Redis is based on PMDK, we also test it with Pmemcheck and observes a 22.3× slowdown (13.6× slower than PMTest). Compared to the previous microbenchmarks, the slowdown is much lower as the real workloads are less intensive in accessing PM. We conclude that PMTest is efficient at testing real workloads.

Workload	Library	Input Client
Memcached	Mnemosyne	Memslap (100k ops/client, 5% set), YCSB (100k ops/client, 50% update)
Redis	PMDK	redis-cli (LRU test, 1M keys)
PMFS (kernel module)	Low-level primitives	NFS (Filebench, 8 clients), MySQL (OLTP-complex, 4 clients)

Table 4. Real workloads from WHISPER benchmark suite [52] (YCSB from [11]).

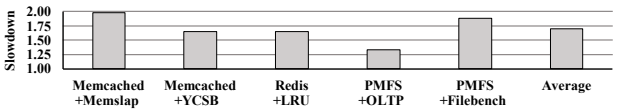


Figure 11. Performance of testing real workloads.

6.2.3 Scalability.

We further analyze the scalability of PMTest using Memcached. We set the number of clients equal to the number of Memcached threads. We manually place checkers to its underlying library, Mnemosyne, to check the consistency of its persistent map. Figure 12a presents the result with variable Memcached threads. As the number of threads in Memcached increases, the slowdown from PMTest increases with both Memslap and YCSB clients due to an increased number of traces generated by the workload. To perform testing more efficiently, we increase the number of PMTest worker threads, as shown in Figure 12b. As the number of workers increases, the slowdown decreases. Then, we increase both the number of workers and Memcached threads at the same time. Figure 12c shows the slowdown slightly increase as both threads increase due to the inter-thread communication overhead. We

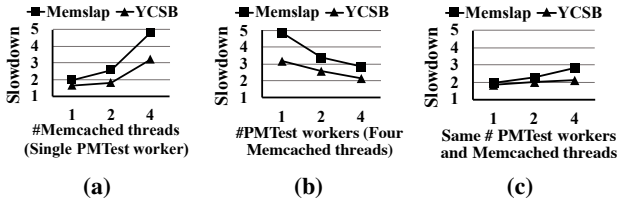


Figure 12. Execution time of Memcached with PMTest.

conclude that PMTest can effectively reduce the testing time when testing PM-operation intensive programs.

6.3 Bug Detection Evaluation

	Bug Type	Description	#Cases	#Checkers
Low-level	Ordering	Missing or misplacement of ordering enforcement	4	18 (Low-level checkers)
	Writeback	Missing or misplacement of the writeback operations	6	
	Performance	Writeback the same persistent object more than once	2	
Transaction	Backup	Missing or misplaced backup of persistent objects	19	2 (Transaction checkers)
	Completion	Incomplete transactions due to improper termination	7	
	Performance	Log the same persistent object more than once	4	

Table 5. Summary of synthetic bugs for PMTest validation.

	File	Line	Description
Known	`xips.c` [26]	207, 262	Flush the same persistent buffer twice
	`files.c` [24]	232	Flush an unmapped buffer
	`rbtree_map.c` [25]	379	Modify a tree node without logging it
New	`journal.c` [27]	632	Flush redundant data when committing
	`btree_map.c` [29]	201	Modify a tree node without logging it
		367	Log the same object twice

Table 6. Summary of the known bugs in the commit history and new bugs detected by PMTest.

To validate the bug detection capability of PMTest, we first systematically create random synthetic bugs in PMDK workloads [33]. Table 5 lists the synthetic bugs we have validated (total 42)[2]. For the programs that uses transactions, we use two pairs of `TX_CHECKER_START` and `TX_CHECKER_END`; for the one uses low-level functions, we place 12 `isPersist()` and 6 `isOrderedBefore()` checkers (the overall benchmark codebase is about 2.6k LOC). PMTest reported all the synthetic bugs we introduced. Then, we reproduced the bugs from the developers' commit history of the workloads that we have previously tested. PMTest also reported these bugs accurately. And finally, during testing, we found three *new bugs* in PMFS and PDMK applications (Table 6). Figure 13 demonstrates the new bugs we have found using PMTest. We simplify the code for readability.

Bug 1 (performance): Figure 13a shows a snippet of code from `journal.c` in PMFS. The function first sets the log entry (`le`) at line 3. Then, it flushes the modified log entry to PM at line 4. Finally, it flushes the entire transaction (`trans`) at line 6. PMTest reports a `WARN` of duplicated flush at line 6. Because the log entry is part of the transaction, the second flush writes back the log entry *again*. A better implementation should flush only the remaining part of the transaction at line 6.

Bug 2 (correctness): Figure 13b shows a snippet of code from `btree_map.c` in PMDK. This function modifies a node

[2]All tested bugs and injected checkers can found at https://pmtest.persistentmemory.org.

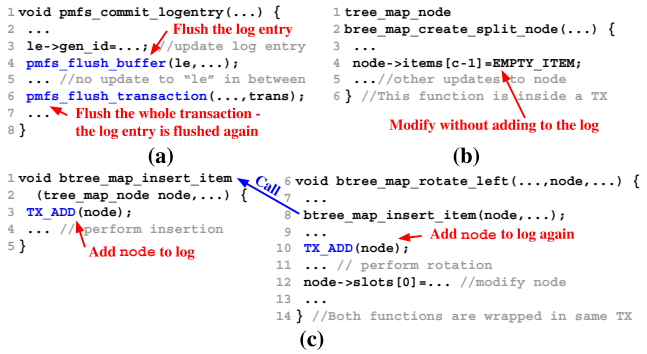


Figure 13. New bugs found in (a) PMFS, and (b, c) PMDK applications.

without logging it. PMTest reports this bug at line 4 and other lines that modify this object. The correct implementation should call `TX_ADD(node)` before line 4. Bug fix from Intel can be found at [31].

Bug 3 (performance): Figure 13c is another snippet of code from `btree_map.c`. The function on the right side first calls the function on the left side and then rotates a tree node. PMTest detects a duplicated `TX_ADD()` at line 10, that should be removed. The function on the left side adds node to the log, while the function on the right side adds the *same* node to the log *again*. As both functions belong to the same transaction, double logging is unnecessary. This bug is subtle as the two log operations are not in the same function. Bug fix from Intel can be found at [30].

We found the two new bugs in PMDK applications using our high-level checkers for PMDK by placing a pair of `TX_CHECKER_START` and `TX_CHECKER_END` around the outermost transaction. We found the bug in PMFS by sending the current trace to the checking engine when the update in `journal.c` commits. The built-in performance-bug checker reports this unnecessary writeback. Therefore, we conclude that using the high-level, automated checkers effectively debugs the program and incurs a minimum effort.

7 Discussion

In this section, we discuss the opportunities and potential issues with using PMTest, and the future works in testing CCS.

7.1 The Use of PMTest

We find out that PMTest can help programmers demystify the semantics of library functions. For example, in a program with nested PMDK transactions (an inner and an outer transaction), we first apply a pair of `TX_CHECKER_START()` and `TX_CHECKER_END()` to the inner transaction. PMTest reports that the updates in the inner transaction are not persisted before the end of the inner `TX_END`. However, all updates to PM are supposed to be persistent when the transaction terminates. Then, we move the checkers to the outer transaction and found that PMTest does not report any bug. Analyzing PMDK source code, we found that updates are guaranteed to be persisted only when the *outermost* transaction ends. PMTest can help programmers check whether library semantics are consistent with what they expect.

7.2 Programmer's Effort using PMTest

Ensuring the crash consistency guarantee relies on two types of correctness: *(i)* algorithmic correctness (e.g., redo/undo logging, checking pointing, etc.), and *(ii)* implementation correctness of that algorithm (e.g., placing the writebacks and fences in the correct place). Even when the programmers use the algorithm of the logging mechanism in a correct manner, the reordering of instructions makes it hard for the programmers to intuitively infer the correctness of the implementation (as shown in Figure 1). Placing the low-level checkers in the code increases the programmer's effort. However, now programmers can assert the expected behavior of the program, and therefore, can ensure the implementation correctness. On the other hand, programmers who use the high-level checkers to test programs (built using the high-level libraries) do not need to understand the low-level algorithm and implementation to ensure crash consistency. Therefore, the high-level checkers minimize programmers' effort. Expert developers of PM libraries can create high-level checkers for their libraries to enable an easy-to-use testing interface for future users of their libraries. This way, ordinary programmers can use those high-level checkers to test their CCS built with high-level libraries.

7.3 Impact of incorrect use of PMTest

The low-level checkers exposed by PMTest work in a similar way as assertions do in conventional programs. Incorrect use of the checkers can cause false alarms and lead the programmer to believe the implementation is incorrect, but will never introduce any new error or bug to the code. In comparison, the high-level checkers require minimal programmers' effort and can mostly be automated. For example, while checking the PMDK library in our evaluation, we only added 9 lines of C code (for initialization, termination, etc.), where the insertion of the high-level checkers were automated. Therefore, we recommend that only the expert programmers use the low-level checkers to avoid any misuse of PMTest interface.

7.4 Future Work

Dynamic v.s. Static Testing of CCS. PMTest takes a dynamic approach that detects crash consistency bugs on the trace that has been executed. This method is limited by the execution path that the program takes based on the input. Therefore, PMTest aims for fast testing in order to cover more input sets. In comparison, static testing methods can overcome the limitation of coverage, while cannot handle issues

related to dynamically allocated memory and pointers. Therefore, static methods tend to set more false alarms compared to dynamic ones. We leave the research on detecting crash consistency bugs statically as a future work.

Testing Multithreaded CCS. In this work, we provided support for multithreaded programs by tracking trace individually on different threads. This support is sufficient for most cases. For example, multithreaded transactions in PMDK are independent as one thread writes back all its persistent data before releasing the lock. WHISPER also shows that inter-thread dependency is rare in persistent programs [52]. We leave debugging crash consistency issues due to improper thread synchronization as a future work.

8 Related Work

In this section, we discuss prior works that provide crash consistency support and test persistent programs.

Mechanisms for Crash Consistency. Prior works have provided a variety of software [2, 4, 5, 7–10, 14, 16, 21–23, 33, 38, 42, 46, 48, 56, 63, 64, 66, 68, 73] and hardware supports [15, 35, 36, 39, 47, 49, 52, 54, 60, 74] to maintain the crash consistency of persistent programs. NV-Heaps [9], Mnemosyne [64], REWIND [5], NVL-C [14], PMDK [33] and LSNVMM [22] provide a software interface to allow programmers to store persistent data on PM in a crash-consistent manner. PMFS [16], BPFS [10], Mojim [73], Strata [42], NOVA [68], NOVA-Fortis [69], and SCMFS [66] provide PM-optimized file systems to store persistent data. PMTest can assist debugging these software-based solutions based on their specifications. DPO [39] and HOPS [52] propose efficient persistency models to allow programmers to maintain crash consistency using low-level functions. Kiln [74], ThyNVM [60], JUSTDO Logging [35] and ATOM [36] provides transactional interface through hardware support to ensure crash consistency. PMTest can test programs built on these hardware-assisted solutions by appropriately extending the checking engine.

Tools for Testing Crash Consistency. There have been tools that test the crash consistency of legacy file systems [6, 18, 19, 41, 51, 61]. For example, Recon [19] adopts a runtime testing approach to test the metadata consistency of Ext3 and Btrfs. However, these tools only work for conventional block devices, such as disks. As PM becomes imminent, recent works provide tools to detect crash consistency issues in PM-based programs. Yat [43] is designed for testing PMFS [16], a PM-optimized file system, using an exhaustive test method. As a result, Yat is extremely slow and cannot be used to test other custom CCS. Pmemcheck [59] and Persistence Inspector [55] are tools designed for testing programs developed with the PMDK library [33]. However, both tools only support the PMDK library under x86 persistency model. PMTest is a more efficient and flexible tool that supports various PM programs and persistency models.

9 Conclusions

In this work, we demonstrate that developing crash consistent software for PM systems is hard and error-prone as the programmers cannot reason about the ordering of persistent operations during the development phase. To this end, we design and implement PMTest, a crash consistency bug testing mechanism that exposes the ordering and durability of the persistent operations to the software. To our knowledge, PMTest is the first tool that is both flexible and fast. We have demonstrated the effectiveness of PMTest by testing CCS developed for PM systems and detected new bugs in user-space applications and in a kernel-space file system. We have also shown that PMTest can be extended to support different persistency models proposed in the literature. We believe that PMTest is highly useful to persistent memory software developers for identifying bugs and understanding the crash consistency guarantees in various types of CCS running on different persistency models.

10 Acknowledgment

We thank the anonymous reviewers, Andy Rudoff and the whole PMDK team at Intel, Korakit Seemakhupt, and Nora Evans for their valuable feedback. This work is supported in part by NFS grants 1829524, 1817077, 1822965, and the SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory (CRISP).

References

[1] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2017.

[2] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2015.

[3] David Brash. Armv8-A architecture evolution. https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-evolution, 2016.

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

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

[6] 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 *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[7] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[8] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2011.

[10] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[12] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly Media, Inc., 3rd edition, 2005.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[14] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. NVL-C: Static analysis techniques for efficient, correct programming of non-volatile main memory systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[15] K. Doshi, E. Giles, and P. Varman. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

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

[17] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[18] Daniel Fryer, Mike Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the integrity of transactional mechanisms. *Trans. Storage*, 10(4):17:1–17:23, October 2014.

[19] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[21] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, 2017.

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

[23] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware logging in transaction systems. *Proc. VLDB Endow.*, 8(4):389–400, December 2014.

[24] Intel Corporation. PMFS: Remove unnecessary flushing from pmfs_fsync(). https://github.com/linux-pmfs/pmfs/commit/e293e14725aaf36d844bfc4a0cb3d4f99fba1f0b, 2013.

[25] Intel Corporation. Add missing undo log entry in rb-tree example (PMDK). https://github.com/pmem/pmdk/commit/04ec84e23ed40be92bd89b9d34c39fbf28cafe0b#diff-f2692f0bb21a212d07a5d1bc2115c071, 2015.

[26] Intel Corporation. PMFS: Remove duplicate flush buffer. https://github.com/snalli/PMFS-new/commit/ded1b075eb911c469233433d83cb678ee800367c, 2015.

[27] Intel Corporation. PMFS. https://github.com/snalli/PMFS-new/blob/2c62f0a20f98afe128e59d5e7f0aff40489b27f7/journal.c, 2016.

[28] Intel Corporation. Intel Xeon Silver 4110 Processor. https://ark.intel.com/products/123547/Intel-Xeon-Silver-4110-Processor-11M-Cache-2_10-GHz, 2017.

[29] Intel Corporation. B-Tree (PMDK). https://github.com/pmem/pmdk/blob/5ac1f5b882275d1eaf6f488a5a71851cb2fdc1ae/src/examples/libpmemobj/tree_map/btree_map.c, 2018.

[30] Intel Corporation. Btree: remove not needed snapshot (PMDK). https://github.com/pmem/pmdk/commit/b9232407a794040102e769ed98b967d797c173fd#diff-f2692f0bb21a212d07a5d1bc2115c071, 2018.

[31] Intel Corporation. Btree: snapshot node before modifying it (PMDK). https://github.com/pmem/pmdk/commit/25f5e4f676e3d9cd7a4c9dc7aa8f2f36e83ff6c2#diff-f2692f0bb21a212d07a5d1bc2115c071, 2018.

[32] Intel Corporation. Intel architecture instruction set extensions programming reference (319433-034 may 2018). https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, 2018.

[33] Intel Corporation. Persistent memory programming. https://pmem.io/, 2018.

[34] Intel Corporation. Revolutionary memory technology. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html, 2018.

[35] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[36] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[37] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.

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

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

[40] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[41] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High-performance metadata integrity protection in the WAFL copy-on-write file system. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST)*, 2017.

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

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

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

[45] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer*

Architecture (ISCA), 2009.

[46] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[47] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[48] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

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

[50] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing legacy code to byte-addressable persistent memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[51] Ashlie Martinez and Vijay Chidambaram. Crashmonkey: A framework to systematically test file-system crash consistency. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[52] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[53] NIST. The economic impacts of inadequate infrastructure for software testing, 2002.

[54] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but No Force: Efficient hardware undo+redo logging for persistent memory systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

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

[56] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[57] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

[58] Persistent Memory Wiki. Persistent memory. https://nvdimm.wiki.kernel.org/, 2018.

[59] PMDK. An introduction to pmemcheck. http://pmem.io/2015/07/17/pmemcheck-basic.html, 2015.

[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 *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.

[61] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[62] Linus Torvalds. https://github.com/torvalds/linux/blob/master/include/linux/kfifo.h, 2013.

[63] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, 2014.

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

[65] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[66] X. Wu and A. L. N. Reddy. SCMFS: A file system for storage class memory. In *SC '11: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[67] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie. Overcoming the challenges of crossbar resistive memory architectures. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

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

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

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

[71] Anna Zaks and Jordan Rose. https://llvm.org/devmtg/2012-11/Zaks-Rose-Checker24Hours.pdf, 2012.

[72] Mingzhe Zhang, King Tin Lam, Xin Yao, and Cho-Li Wang. SIMPO: A scalable in-memory persistent object framework using NVRAM for reliable big data computing. *ACM Trans. Archit. Code Optim.*, 15(1):7:1–7:28, March 2018.

[73] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[74] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.