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

1 Introduction

Persistent Memory (PM) technologies, such as Intel's Optane DC PM [6], offer the persistence of disks combined with performance close to that of DRAM. PM enables software to directly manage persistent data, bypassing OS indirections, significantly improving the performance over conventional systems. Various software systems are being developed and deployed to leverage the benefit from PM. Examples include PM-optimized file systems [2, 13], databases [9], and many other customized applications and libraries [1, 5, 12]. The persistent data on PM is expected to be recoverable in the event of a crash (e.g., power failure). We refer to this requirement as the crash consistency guarantee, where the persistent data managed by the program is always in a consistent state.

1.1 Background in Persistence Programming

Programming in PM systems for crash consistency is hard and error-prone. Properly implemented crash consistent software needs to meet two fundamental requirements: *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 the update has become persistent. As a solution, the x86 ISA introduced new optimized instructions (e.g., `CLWB`) to efficiently write back cache lines to memory, and the ARM ISA introduced the `DC CVAP` instruction to write back a cache line to the persistence domain. We refer to the act of making a cache line persistent as a *persist operation*. Enforcing ordering is another fundamental necessity to ensure crash consistency. An *ordering guarantee* from the PM system is required 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 requires the undo log entry to be created and persisted *before* the associated data get modified. The x86 ISA provides ordering guarantees through the `SFENCE` instruction. A 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 writes. For simplicity, we refer to the combination of "`CLWB;SFENCE`" as a `persist_barrier`. When developing crash consistent software for PM systems, programmers must carefully use these low-level primitives for correctness. Next, we use an example to demonstrate the difficulty in programming for PM.

Example. Figure 1 shows a snippet of code that updates an array on PM 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. 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 the change of the valid bit (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`: one right after the creation of the backup copy (between lines 2 and 3), and another right after updating the new array index (between lines 5 and 6). Omitting either 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 update to the array has not. Therefore, after recovery, the array will treat the stale value in memory as the updated one.

```
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();
5   array[index] = new_val;    //Update to the new value
6   backup.valid = false;      //Set the backup as invalid
7   persist_barrier();
8 }
```

Missing `persist_barrier()`

Figure 1. Example of a buggy PM program.

Even with the help of transactional libraries that build upon these low-level primitives [1, 5, 12], 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 specified order will *not* result in a runtime ordering that *violates* the required *persist* order. We refer to the bugs that cause failure of recovery as *crash consistency bugs*.

1.2 Challenges in Testing Crash Consistency

We argue that PM software developers will greatly benefit from a testing framework that can help identify the improper use of low-level primitives or high-level libraries, however, there are two major challenges in developing such framework. First, while prior works have developed tools to assist PM software development, they are all specific to certain file systems [8] or user-space libraries [11]. 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 [8], a tool that tests Intel's persistent memory file system (PMFS [2]) 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 PM software 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 use a strict ordering of writes, while recent academic proposals relax this ordering [7, 10]. In this work, we propose PMTest, a crash consistency testing framework that is, unlike prior work, both *flexible* and *fast*.

2 PMTest

2.1 Key Ideas

We implement PMTest on top of the following key ideas to overcome these challenges.

Flexible. Our key idea is based on the observation that regardless of the difference in PM software (kernel modules, or custom applications using architecture-specific low-level primitives or high-level libraries), they all fundamentally rely on two types of operations to

provide the durability and ordering guarantees: to enforce persisting a write and to enforce ordering between writes. Accordingly, we propose two low-level *checkers* for testing crash consistency of PM software: `isPersist()` and `isOrderedBefore()`, that check (1) whether certain persistent objects have been persisted since their last update and (2) whether a certain *persist* operation has been ordered before another, enabling testing of the two fundamental properties of any crash consistent PM software. Similar to assertions, these two checkers can be instrumented in the code to expose the ordering and durability of persistent operations to the software. Moving forward, 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. High-level checkers can automate the process of debugging PM software built with PM libraries. We have developed high-level checkers compatible with transactions in Intel's PMDK [5] and an academic work Mnemosyne [12]. Programmers only need to place a pair of `PMTest_START` and `PMTest_END` around the transaction without requiring any understanding about program implementation.

Fast. An effective testing tool is required to capture all interleavings among persist operations, as writes may become persistent in an order different from the original program order. The existing method [8] exhaustively tests the correctness of recovery under all possible interleavings. Instead of permuting all cases, PMTest deduces an *interval* in which a write can possibly become persistent based on a trace of PM operations (e.g., write, cache writeback and fence). We refer to this interval as a *persistence interval*. An overlapping persistence interval for two write operations implies that they are *not* strictly ordered, and the ending time of the interval determines at what point in the program the write is guaranteed to persist. Therefore, the persistence interval naturally captures all reordering possibilities. Based on the persistence interval, PMTest determines whether the conditions specified by the aforementioned checkers (both low-level and high-level) are met, and thus detects crash consistency bugs.

2.2 Implementation

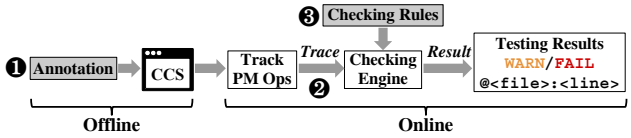

Figure 2. A high-level view of PMTest (shaded components can be customized by programmers).

Figure 2 briefly describes the high-level workflow of PMTest. The procedure of testing a program consists of *offline* and *online* steps. In the *offline* step, programmers annotate the software using low-level and/or high-level checkers following the program specification of the crash consistency mechanism (step ❶). For example, the low-level checkers should be inserted to check the programmer intended crash-consistent behavior and the high-level checkers for transactions can be added by wrapping up the transactions. In the *online* step, PMTest executes with the annotated and compiled software. During execution, PMTest tracks PM operations and passes the trace to the checking engine (step ❷). PMTest supports both user-space programs and kernel modules by running the tracking module and testing engine in separate processes. The checking engine tests whether the trace meets the requirements specified by the checkers (step ❸). Upon identifying a crash consistency bug, PMTest reports the buggy file name and line number for debugging. PMTest comes with the checking rules for x86 systems and is extensible to other hardware platforms.

3 Evaluation

We evaluate PMTest in two dimensions: (1) its capability of detecting crash consistency bugs and (2) its runtime overhead in testing real-world workloads.

Testing Capability. We evaluate the capability of PMTest bug detection in two ways. First, PMTest detected 45 manually created bugs (synthetic and reproduced from the commit history) in WHISPER [10], a benchmark suite for PM. Second, PMTest detected 3 new bugs in a file system (PMFS) and applications developed using a transactional library (PMDK). These bugs have been reported to Intel and have been fixed with proper credit to PMTest [3, 4]. Further, our experiments also reveal that PMTest checkers can help programmers understand the persistency guarantees of PM libraries.

Testing Performance. We evaluate three real workloads: two PM-optimized databases, Redis and Memcached, and a PM-optimized file system, PMFS. Figure 3 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×. We conclude that PMTest is efficient at testing real workloads.

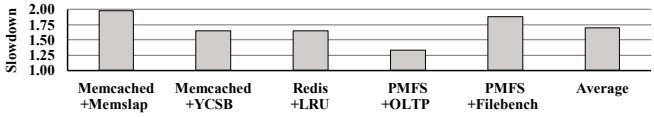

Figure 3. Performance of PMTest.

4 Conclusions

Developing crash consistent PM software is hard and error-prone. As both the hardware and software support for PM is being developed, programming for PM software will be facing a variation in hardware platforms and software libraries. Among different software and hardware systems, the ordering and durability guarantees may vary. Therefore, programmers are required to be experts on both PM software and hardware systems. To the best of our knowledge, PMTest is the first work that enables testing of PM software ranging from user-space programs to kernel modules. And, PMTest, as a testing framework, has reserved the extensibility for future PM hardware and libraries. On the other hand, PMTest minimizes runtime overhead in testing. Compared to Intel's existing testing tool, Pmemcheck [11], PMTest provides better testing capabilities while performs 7.1× faster when testing workloads built on the PMDK library. Using PMTest, we have detected 3 new bugs in existing PM software, including a PM-optimized file system (PMFS) and applications based on a transactional library in PMDK.

References

[1] Coburn et al. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.
[2] Dulloor et al. System software for persistent memory. In *EuroSys*, 2014.
[3] Intel. Btree: remove not needed snapshot (PMDK). https://bit.ly/367Jc1m, 2018.
[4] Intel. Btree: snapshot node before modifying it (PMDK). https://bit.ly/2BLZHCo, 2018.
[5] Intel. Persistent memory programming. https://pmem.io/, 2018.
[6] Intel. Revolutionary memory technology. https://intel.ly/31PvSLw, 2018.
[7] Kolli et al. Delegated persist ordering. In *MICRO*, 2016.
[8] Lantz et al. Yat: A validation framework for persistent memory software. In *ATC*, 2014.
[9] Marathe et al. Persistent Memcached: Bringing legacy code to byte-addressable persistent memory. In *HotStorage*, 2017.
[10] Nalli et al. An analysis of persistent memory use with WHISPER. In *ASPLOS*, 2017.
[11] PMDK. An introduction to pmemcheck. http://pmem.io/2015/07/17/pmemcheck-basic.html, 2015.
[12] Volos et al. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.
[13] Xu et al. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, 2016.