A Holistic System Support for Persistent Memory

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Sihang Liu

July 2022

Acknowledgments

My PhD career is full of challenges. The support from my parents, advisor, mentors, collaborators, and friends helped me go through this process. First of all, I would like to thank my parents, Jianfeng Liu and Jiwen Shi, who have been very supportive during my PhD career. I grew up in a family full of engineers and scientists. My father, Jianfeng Liu, and my grandparents Linbiao Shi and Yifang Li, encouraged me to explore engineering when I was a child. And my mother, Jiwen Shi, a medical technician, introduced scientific research to me when I was in elementary school. I really appreciate that I was exposed to engineering and scientific research in my childhood.

Next, I would like to thank my advisor, Samira Khan, who has been extremely supportive during my PhD career. During the early stage of my PhD, Samira tightly worked with me through brainstorming, experimentation, paper writing, and presentation. Beyond the technical aspects, Samira is also very supportive of my career, including award applications and the academic job search. It is impossible for me to receive a PhD degree without her advice and support.

I would also like to thank my mentors who constantly give me support and advice. I would like to thank Valeria Bertacco, who hosted my summer research internship—my first experience in computer architecture research—when I was an undergraduate student at the University of Michigan. After that, I was exposed to persistent memory research—a direction I took throughout my PhD—during my research internship in Tom Wenisch's lab at the University of Michigan. The research experience at Tom's lab has inspired me to pursue a PhD degree in the area of computer systems and architecture. After joining the University of Virginia as a PhD student, Tom still gave me valuable advice, both on my research and career development. During my visit at Tom's lab, Aasheesh Kolli was my mentor, who has been supporting my research not only when I was at Michigan but also after I joined Virginia. Hassan Wassel was my mentor from the Google PhD Fellowship Program. This mentorship helped me better understand real-world research problems in the industry and build my research vision. I would also like to thank Gennady Pekhimenko, Baishakhi Ray, Kevin Sakdron, Mircea Stan, Steven Swanson, Yuan Tian, and Jishen Zhao, who provided me with valuable advice on both my research and academic job application.

During my PhD, I was fortunate to work closely with a number of collaborators. I would like to thank Kevin Angstadt, Rachata Ausavarungnirun, Jianfeng Chi, Amel Fatima, Michael Ferdman, Daniel Gruss, Zichao Hu, Suraaj Kanniwadi, Yasunari Kato, Samira Khan, Andreas Kogler, Aasheesh Kolli, Suyash Mahar, Gennady Pekhimenko, Han Jie Qiu, Baishakhi Ray, Jinglei Ren, Tajana Rosing, Martin Schwarzl, Korakit Seemakhput, Yasas Senvenati, Muhammad Shahbaz, Faysal Shezan, Yun Joon Soh, Kevin Song, Steven Swanson, Yuan Tian, Hassan Wassel, Yizhou Wei, Tom Wenisch, Vinson Young, Jishen Zhao, Hanzhi Zhou, and Minxuan Zhou.

Among my collaborators, I would like to additionally thank my labmates in ShiftLab at the University of Virginia: Amel Fatima, Suraaj Kanniwadi, Yasunari Kato, Suyash Mahar, Korakit Seemakhput, Yasas Senvenati, and Yizhou Wei. I still remember the good old days when I worked closely with them till midnight, before conference deadlines.

Besides research, I am also fortunate to work with a group of people from the Computer Architecture Student Association (CASA) during my PhD. First, I would like to thank Emily Ruppel, the co-chair and a co-founder of the Computer Architecture Long-term mentoring (CALM), a program affiliated with CASA. And, I would like to thank other co-founders of CALM, Elba Garza, Yueying Li, Suyash Mahar, Abdulrahman Mahmoud, Gururaj Saileshwar, and Annus Zulfiqar, who helped make this mentoring program possible. Besides, I would also like to thank Joel Emer, Saugata Ghose, and Talia Ringer, who have provided precious advice to CALM.

My PhD career is a six-year journey. I was fortunate to be accompanied by friends who I could share my happiness and sadness with, especially during the quarantine period due to COVID-19. I would like to thank Weilin Xu, Aihua Chen, Suya, Xida Ren, and Jianfeng Chi for the dinners and picnics; Zhongzheng Tian, Yang Yang, Jiayu Zhou, and Farzana Ahmed Siddique for the fantastic hiking experience in Virginia; Yi Jiang for introducing photography to me, a hobby that helped me stay positive during the pandemic days; Sergiu Mosanu, Tom Tracy, Vaibhav Verma for the great time in the bar after work; Dengwang Tang and Ye Fei for hosting me during my trip back to Ann Arbor; Yi Zhang for showing me around Princeton; Zixuan Wang for showing me the campus of UCSD during my visit at San Diego.

Abstract

Persistent memory (PM) technologies, such as Intel's Optane memory, unify memory and storage and deliver both data persistence and high-performance. PM systems allow programs to directly manage their persistent data in memory, as opposed to the conventional way that goes through the file system. Though performant, integrating this new memory technology would require significant changes throughout the system stack. First, programs that directly manage persistent data need to guarantee data recovery after a failure, as the file system is bypassed. However, it is hard and error-prone to ensure failure-recovery as programs need to carefully manage writes to PM. Second, PM is both a memory and a storage device, which requires various memory and storage supports, such as memory encryption and integrity verification that secure the data and memory deduplication for better bandwidth. Among these supports, the security guarantees are critical but can significantly increase the access latency. Moreover, these supports should also follow the existing crash consistency guarantees. Third, even with data encryption and integrity verification, there can be other vulnerabilities in a real PM system. For example, Intel's Optane PM uses multiple levels of caches and buffers to improve performance, which can lead to new side channels.

My thesis aims to provide system supports to overcome these new challenges. We hypothesize that a whole-system-level redesign, from programming support to hardware, that ensures correctness, security, and high-performance, is necessary in order to integrate persistent memory into practical systems. On the software side, to ensure the failure-recovery correctness, we have developed testing tools, PMTest and XFDetector, to help programmers detect failure-recovery issues; and a test case generator, PMFuzz, to generate high-coverage test cases. On the hardware side, we have proposed efficient and secure hardware-software co-designs for PM systems. Further on, we have reverse-engineered the commercial Optane PM from Intel, and exploited its covert and side-channel vulnerabilities.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Sihang Liu

This dissertation has been read and approved by the Examining Committee:

Samira Khan, Advisor

Kevin Skadron, Chair

Baishakhi Ray

Mircea Stan

Yuan Tian

Thomas Wenisch

Accepted for the School of Engineering and Applied Science:

Engineering Dean, Dean, School of Engineering and Applied Science

July 2022

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The advancement in memory technologies brings a new type of memory, persistent memory (PM) or non-volatile memory (NVM), that offers data persistence similar to storage devices, such as Solid State Drives (SSDs), while delivering performance close to dynamic random-access memory (DRAM). With Intel's release of Optane DC Persistent Memory [5], this new class of memory technology has finally become available to the industry, researchers, and developers. There are also other PM technologies such as ReRAM [6] and STT-RAM [7] under research that promise better performance. These PM devices are placed on the memory bus, alongside DRAM, and can be accessed through a byte-addressable load/store interface. Therefore, the adoption of PM blurs the boundary between storage and memory, moving the persistent domain from storage to the main memory. This unification of storage and memory brings a significant performance improvement to applications that manage and manipulate persistent data, such as file systems [8–11] and databases [12–16].

The benefit from adopting PM is prominent, however, accessing persistent data through a level of indirection in the file systems or databases overshadows the benefit of this fast memory technology. As PM enables direct access to the persistent storage at a fine granularity, programs can bypass such indirection and manage their persistent data through a direct, byte-addressable load/store memory interface. However, the opportunity of achieving better performance by managing persistent data directly also brings new challenges to the system design. The *first challenge* is in the development of correct programs for the new PM system. One major requirement of persistent data is their recoverability to a consistent state in event of a system crash (e.g., unexpected power outage and system crash). This requirement is also known as the *crash consistency guarantee*. In a conventional system, programs access persistent data through file systems that manage data recovery after power cycles or unexpected failures. In comparison, PM systems provide direct access capabilities to move the file system overheads off the critical path. Although this mode provides better performance, programs need to directly manage persistent data and leave the burden of maintaining their crash consistency to programmers, which is difficult and error-prone. With the release and integration of Intel's DC Persistent Memory, there is an urgent need for assisting programmers to develop PM-optimized programs and ensure their crash consistency guarantee.

The *second challenge* is to design secure and efficient PM hardware systems. As PM is not only a type of memory but also a storage device, it needs system support that has been applied to storage devices, such as encryption, integrity verification, wear-leveling, deduplication, and compression. These various kinds of system support ensure security and reliability, and optimize the bandwidth and capacity of PM. These system supports should be low-overhead and work with the PM-based programs seamlessly. However, a naive integration of these supports into PM hardware can lead to a degradation in performance. For example, integrity verification can lead to hundreds of nanoseconds of extra latency to memory accesses. Even worse, additional memory operations in these memory and storage supports can break the crash consistency guarantees. For example, each data block in a counter-mode encryption system [17] requires an additional counter for encryption. If the versions of data and the counter do not match, the data block will fail to decrypt. Existing PM hardware systems, however, do not provide such a guarantee. To solve these issues, a redesign of PM hardware system stack will be necessary.

The *third challenge* is in the potential side-channel vulnerabilities in PM devices. The PM storage media itself, such as Intel's Optane technology, cannot deliver the DRAM-like performance through the DDR interface directly. To bridge the performance gap between the PM storage media and the DDR memory interface, the commercial PM device introduces buffers and caches, as well as internal memory remappings [18–22]. Prior works on hardware side-channel attacks, such as Spectre [23] and Meltdown [24], have exploited transient execution in the processor to break the existing isolation provided by the operating system. As the Optane PM is a complicated system, there can be new security concerns. Thus, there is a need to study and mitigate the potential vulnerabilities before attackers can exploit them in real life.

Thesis Statement: We hypothesize that a whole-system-level redesign, from programming support to hardware, that ensures correctness, security, and high-performance, is necessary in order to integrate persistent memory into practical systems.

1.1 Theme 1: Correctness guarantees for PM programming

The performance benefit of PM comes from not only its faster storage media but also the direct access to persistent data, without going through file system indirections. However, the programs are still expected to be recoverable in event of a failure. To meet the requirements for recovery, the programs need to integrate failure-recovery mechanisms such as redo/undo logging, shadow paging, and checkpointing, by carefully managing writes to PM. Therefore, programming for PM is hard and error-prone. We refer to programming errors that cause programs to fail in recovery as *crash consistency bugs*. This research theme aims to assist programming for PM systems.

In Chapter 3A, we introduce our testing tool, PMTest [25], that exposes the low-level and persistence of PM writes to programmers, and then checks the ordering and persistence states against a specification. Violations of the specified ordering and persistence requirements indicate the implementation is buggy and cannot lead to a consistent recovery.

Further, we find that recovery correctness not only depends on the normal execution before failure but also the recovery procedure that restores data. If either stage fails, the program can end up in a non-recoverable state. In Chapter 3B, we present XFDetector [26], a testing tool that increases the testing scope from program's normal execution phase to the whole duration of program's execution. By placing failures in the program and attempting to recover from the failure, XFDetector tests both the normal execution stage and the recovery procedure end-to-end.

PMTest and XFDetector make it possible to detect crash consistency bugs. However, detecting these bugs sometimes requires strong tests. For example, a procedure during recovery may require a certain state on the PM image to trigger. Therefore, in Chapter 3C, we introduce PMFuzz [27], a test case generator for covering program paths that can lead to crash consistency bugs.

1.2 Theme 2: Secured and high-Performance PM systems

Different from conventional memory systems, PM is not only a type of memory but also a storage device. Therefore, a practical PM system would require supports that guarantee security and reliability and optimize for capacity and bandwidth. For example, encryption and integrity verification guarantee security, compression and deduplication improve PM bandwidth, and wear-leveling extends PM's lifetime and reliability. This theme focuses on the security guarantee as it is one of the most important aspects. Counter-mode encryption is a commonly used memory encryption method, where a random counter value is encrypted and then XORed with the plaintext to generate an encrypted block. Therefore, during decryption, the counter value and the encrypted data block need to match. However, simply integrating encryption into the PM system can break the crash consistency guarantees. If an encrypted data block is not written atomically with its counter, they can mismatch in case of a failure. In Chapter 4A, we propose an efficient design that ensures crash consistency in an encrypted PM system, which we refer to as counter atomicity [28]. Counter atomicity ensures that the encrypted data block and the counter always become persistent in PM at the same time, i.e., atomically. To reduce the atomicity overhead, we further relax the requirements by selectively applying counter atomicity to writes that immediately affect the crash consistency guarantees in a program, which we refer to as selective counter atomicity.

Besides the impact on recoverability, we find that these memory and storage support, including the aforementioned encryption operation, can increase the write latency. For example, integrity verification can take hundreds of nanoseconds [29]. As crash consistency mechanisms in PM programs add additional ordering constraints to writes, the write latency is placed on the critical path. To optimize the increased write latency due to these memory and storage operations, we propose Janus [30] in Chapter 4B. Janus is a software-hardware co-design that takes two key methods to optimize the write latency. First, in Janus, memory and storage operations are divided into smaller steps such that they can be executed in parallel. Second, Janus provides a software interface that allows for these operations to be executed ahead of time, before the write happens, as soon as they have their address and/or data dependencies resolved. For example, in a key-value insertion function, the data dependency of the value update is known at the function call, and its address dependency is known as soon as after lookup.

1.3 Theme 3: Side-channel vulnerabilities in PM hardware

Although data on PM can be secured using encryption and integrity verification, there can be other vulnerabilities in real PM systems. Prior works [18–22] have suggested that the commercially-available PM, Intel's Optane DC Persistent Memory (DCPMM) [5] is not a monolithic device;

instead, it contains buffers, caches, and wear-leveling mechanisms for better performance and endurance. Therefore, attackers can leverage these hardware structures in the Optane DIMM to establish covert channels and perform side-channel attacks to break the existing system-level isolation.

To assess the hardware side-channel vulnerabilities, in Chapter 5, we first reverse-engineer the Optane module to have a more thorough understanding of its internal hardware structures, including the cache associativity and replacement policy, and the wear-leveling mechanism. Then, we build attack primitives based on the reverse-engineered designs. Overall, we provide four attack primitives: (1) Optane's Read-Modify-Write (RMW) buffer, (2) the Address-Indirection-Translation (AIT) buffer, (3) read-write contentions, and (4) the wear-leveling event that can lead to significant latency increase. Then, we demonstrate four attacks and covert channels based on primitive 1–3: a keystroke attack that monitors user's inputs using primitive 1, a remote covert channel based on the wear-leveling latency, and a remote, asynchronous covert channel using the persistence property of the wear-leveling mechanism.

1.4 Summary

The thesis is organized as the following. First, Chapter 2 introduces the background on PM systems, covering both the software, hardware, and security aspects. In Chapter 3, we will first talk about two testing frameworks that guarantee correct persistence and ordering, and end-to-end correctness with consideration for the recovery procedure. Then, we will describe a test case generator that enables efficient testing. In Chapter 4, we will talk about software-hardware co-designs that guarantee efficiency and crash consistency in PM hardware systems that integrates memory and storage operations for security, endurance, and bandwidth optimization. In Chapter 5, we further assess the side-channel vulnerability of the existing Optane PM from Intel, and demonstrate side-channel attacks and covert channels in Optane-based PM systems. Finally, Chapter 7 summarizes the thesis.

Chapter 2

Background

In this chapter, we provide the necessary background to understand the rest of the thesis. We first introduce both experimental and commercial PM technologies (Section 2.1). Then, we introduce the existing software (Section 2.2) and hardware support (Section 2.3) for PM that enables the recovery of persistent data in case of a failure and meets practical requirements for security, endurance, and memory bandwidth. Finally, we describe the security implications of the persistent memory hardware (Section 2.4).

2.1 Persistent Memory Technologies

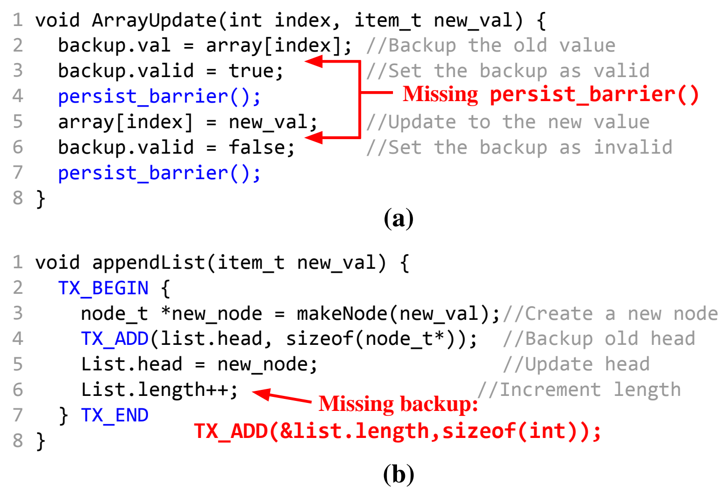
Persistent memory (PM) technologies are a new class of memory technology that aims to deliver high performance and byte-addressability, similar to DRAM, and at the same time, maintain data persistence, similar to hard drives. There have been many proposals on PM technologies. For example, the phase-change memory (PCM) [31] places a phase-change material in between two electrodes, which can be programmed to a high-conductive and a low-conductive phase to represent bit values of 1 and 0. The resistive random-access memory (ReRAM) [6] is similar to PCM except that it uses a metal oxide layer to control the resistance and record 0s and 1s. ReRAM cells can also work as multi-level cells (MLCs) to store multiple bits by further differentiating the resistance levels [32] (e.g., four resistance levels can represent two bits). The spin-transfer torque random-access memory (STT-RAM) [7] uses the spin of electrons to present different values. These technologies are mainly in an experimental stage, whereas Intel has already released its Optane PM [5] that uses their 3D XPoint technology. A system with Optane can have tens of TBs of PM installed on the memory bus, alongside the DRAM modules. Compared to conventional storage devices, such as SSDs and HDDs, Optane PM allows programs to better leverage the low access latency by directly managing persistent data in PM. A common approach is to perform the direct access is to create a PM image in a file system with the direct access support (e.g., Ext4-DAX), map it to the program's address space, and manipulate the persistent data with loads and stores [33]. This way, programs can bypass the OS indirections, such as the file system, to shorten the data path. Next, we take a further look into the software systems for PM.

2.2 Software Systems for Persistent Memory

The elimination of file system indirections improves performance, but introduces additional requirements to the programs. PM-optimized programs need to maintain persistent data in a recoverable state in case of a failure, which we refer to as the *crash consistency guarantee*. Due to the reordering and buffering in the memory hierarchy, the order a write becomes persistent may differ from the program order. For example, if the program performs two writes to location `A` and `B` on PM, where the write to `A` happens before the one to `B` in program order. Without enforcing the persist ordering, the write to `B` can become persistent before the one to `A`, as cache replacement and memory reordering are not under program's control. To support programming for PM systems, hardware platforms have introduced instructions to maintain a correct persist ordering. For example, in an x86 system, a sequence of "`CLWB;SFENCE`" instructions [3] ensures that a cache line will be persisted prior to subsequent writes (referred to as a `persist_barrier()`); in an ARM system, similar functionalities can be implemented using a sequence of "`DC CVAP;DSB`" instructions [34].

2.2.1 Programming with Low-Level Primitives

These primitive operations make it possible to implement crash-consistent programs for PM systems. However, using these primitives directly is difficult as the programmers need to understand the hardware platforms and carefully manage writes at a fine granularity. We provide a simple example to show the difficulty associated with using these low-level primitives. Figure 2.1a 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



```
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 }
```

```
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
5     List.head = new_node;                //Update head
6     List.length++;                      //Increment length
7   } TX_END
8 }
```

Figure 2.1: Buggy examples using (a) low-level functions and (b) a transactional interface.

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 anyone 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.

2.2.2 Programming with PM Libraries

To simply the programming effort, there have been works that develop PM libraries and provide higher-level software interfaces, such as transactions and persistent data structures for better programmability. For example, Intel's PMDK library [35] provides a transaction interface. The procedure within a PM transaction is expected to be failure-atomic, i.e., either recover to a state before the transaction has happened or complete the transaction. There are also libraries from academia, such as Mnemosyne [36], NV-Heaps [37], and MOD [38]. These libraries are built upon the hardware primitives but abstract away the low-level details for easy programming. For example, with the transactional interface from PMDK [35], 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. Next, we demonstrate another example that uses the PMDK transaction interface.

Figure 2.1b shows an insertion function of a linked list 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, if a failure happens after the `length` has been incremented but before the transaction completes (between line 6 and 7), the program will not be able to recover the correct length of the linked 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.

2.2.3 Crash Consistency Bugs

In the example of Figure 2.1a, the programmer intended to set/unset the valid bit *after* persisting the backup/update, but misses the `persist_barrier`s. Similarly, in the example of Figure 2.1b, the programmer intended 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 crash-consistent software 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. To assist programming for PM systems, in Chapter 3A and 3B, we introduce two testing frameworks, PMTest [25] and XFDetector [26] that expose the hard-to-detect crash consistency bugs. For more efficient testing, in Chapter 3C, we further demonstrate a test case generator, PMFuzz [27], that generates test cases for PM programs.

2.3 Hardware Systems for Persistent Memory

In this section, we introduce PM hardware systems that not only enable crash consistency guarantees but also meet practical requirements for security, endurance, and memory bandwidth.

2.3.1 Crash consistency support from the PM hardware systems.

The examples we have demonstrated so far are based on the existing platform that uses Intel's Optane PM. To achieve better performance, there have also been research proposals that introduce new hardware features. One approach is to design a more efficient persistency model [39] to reduce the overhead of persisting data. For example, DPO [40], HOPS [1], PMEM-Spec [41], and Themis [42] propose more efficient persistency models over Intel's system. On the other hand, there are also hardware systems that ensure crash consistency through hardware extensions. Intel processors have provided Asynchronous DRAM Refresh (ADR) [43] that lifts up the persistence domain to the write queue in the memory controller to reduce the latency of data persistence. Beyond the write queue, Intel recently introduced an extended Asynchronous DRAM Reference (eADR) scheme [44] that further lifts up the persistence domain all the way to the caches, eliminating the need for cache write-back/flush operations. There are also proposes from academia. For example, Kiln [45] and ATOM [46] propose hardware-based crash-consistent transactions by integrating persistent caches to hold temporal data during a transaction; in case of a failure, persistent data in the cache can still recover the in-flight transactions. ThyNVM [47] on the other hand, is a hardware-based checkpointing mechanism that saves the volatile processor and memory state to PM to enable resumption after system failures. These hardware proposals minimize the programming effort and reduce the overhead in maintaining crash consistency.

In summary, these hardware systems for PM enable crash consistency, either through primitive operations that apply to writes or more transparent, specialized hardware extensions. However, besides crash consistency, the PM hardware also needs to make sure the data is secure to protect from attackers who have physical access to the PM device.

2.3.2 Encryption in PM Hardware

A common approach to secure the data is to encrypt/decrypt data on every memory access using an encryption engine located in PM controller. Unfortunately, memory reads are on the critical path of the program execution. Thus, the additional latency to decrypt data can significantly degrade

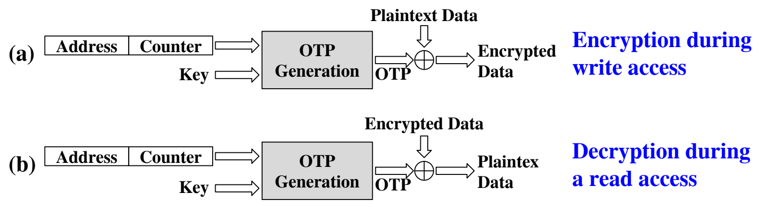


Figure 2.2: The counter-mode encryption technique: (a) encrypting data during a write access, and (b) decrypting data during a read access.

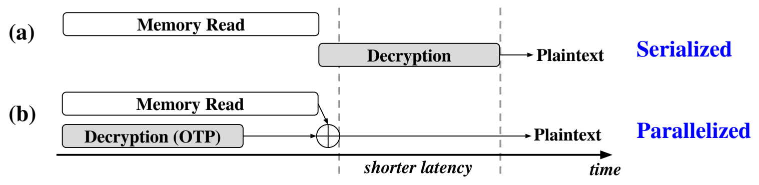


Figure 2.3: Reduction in latency with the counter-mode encryption technique during a read access.

the overall performance. To hide the decryption latency, prior works propose to use the *counter-mode encryption* that makes it possible to parallelize the read access and decryption of data in PM systems [17, 48–51]. In this technique, data is *not* directly encrypted, instead, a unique counter associated with each write access is encrypted to generate a bit string called *one-time-padding* (OTP) (shown in Equation 2.1). This OTP is XORed with the plaintext data to generate the encrypted data (shown in Equation 2.2, Figure 2.2a). As a result, during a memory read access, the OTP is generated using the associated counter while data is still being fetched from PM. When the read access completes, the encryption engine XORs this OTP with the fetched encrypted data to generate the plaintext (shown in Equation 2.3, Figure 2.2b).

$$OTP = En(address|counter, key) \tag{2.1}$$

$$EncryptedCacheline = OTP \oplus plaintext \tag{2.2}$$

$$plaintext = OTP \oplus EncryptedCacheline \tag{2.3}$$

As counters are required to encrypt and decrypt data for *all* memory accesses, the counters are buffered on-chip in a *counter cache* [52], such that the encryption engine does not need to perform an extra memory read access to fetch the counter value. Figure 2.3a shows the serialized decryption technique that adds additional latency to read accesses and Figure 2.3b shows that the read access is faster with the counter-mode encryption technique as the read access and decryption can be performed in parallel.

The main problem with providing crash consistency for an encrypted PM system is that each encrypted data is associated with a counter in the counter-mode encryption, but this relationship is

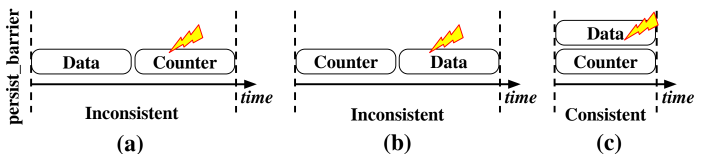


Figure 2.4: (a) Inconsistent decryption if counter write fails, (b) Inconsistent decryption if data write fails, and (c) Consistent decryption if data and counter writes are atomic.

not exposed to the crash consistency mechanisms. While decrypting a cache line after a crash, the memory controller assumes that each memory address has its latest counter in PM. However, decryption will fail if the versions of data and counter are not in sync (either data or counter in PM is stale). Figure 2.4 demonstrates that a system failure can result in out-of-sync data and counter. Every write access to PM consists of two separate write requests, one for the encrypted data and the other for the counter. If a system failure occurs after the data write reaches PM and before the counter write does, the memory controller would observe a stale counter value upon system recovery, introducing inconsistency in data recovery, as shown in Figure 2.4a. Similar inconsistency occurs if a failure happens after the counter reaches PM but the data has not yet been persistent, as shown in Figure 2.4b. As $OriginalVal = En(address|counter, key) \oplus EncryptedVal$, then decryption failure happens in these two cases:

$$\begin{aligned} En(address|counter_{stale}, key) \oplus EncryptedVal_{new} \neq OriginalVal \\ En(address|counter_{new}, key) \oplus EncryptedVal_{stale} \neq OriginalVal \end{aligned} \tag{2.4}$$

Therefore, to enable correct encryption in PM systems, it is necessary to ensure consistency between the data and its associated counter. In Chapter 4A, we describe a design that ensures this consistency by ensuring the data and its counter are updated atomically, as shown in Figure 2.4c.

2.3.3 Other Memory and Storage Support for PM Hardware Systems

So far, we have introduced the background on encryption in PM systems. Besides encryption that ensures data confidentiality, there are also other requirements in practical PM systems. Attackers can also tamper with the data on PM. Thus, to ensure the integrity of data, recent works also use integrity verification techniques [57–59]. Moreover, PM has a limited lifetime [31, 94]. A practical PM system needs to overcome the limitation in lifetime. Prior works have proposed wear-leveling [54, 92, 93] and error correction [88–90] techniques to mitigate the lifetime issue. PM also has a limited write bandwidth compared to that of read [95–97]. A common way of overcoming the write bandwidth is to reduce the write traffic using compression [78–87, 98, 99] or deduplication [73–77] techniques. We summarize the existing flavors of memory and storage support for PM in Table 2.1. These memory

Table 2.1: A description of the existing backend memory operations in PM systems.

Type	Backend Operation	Description	Extra Write Latency
Security	Encryption [28, 48–50, 53–59]	Ensures data confidentiality. Counter-mode encryption is typically used in PM.	40 ns [28, 48]
	Integrity Verification [29, 48, 57–64]	Ensures the integrity of data preventing unauthorized modification. Typically, a Merkle Tree (a hash tree) is used to verify memory integrity.	360 ns (assume a 9-layer Merkle Tree) [48]
	ORAM [65–72]	Hides the memory access pattern by changing the location of data after every access.	$\sim$ 1000 ns [67]
Bandwidth	Deduplication [73–77]	Reduce write accesses that have duplicated data to reduce the write bandwidth.	91–321 ns [77]
	Compression [78–87]	Reduce the size of memory accesses to save the bandwidth.	5–30 ns [83, 85]
Endurance	Error Correction [88–90]	Corrects memory error. Typical solutions include error-correcting code and pointers.	0.4–3 ns [91]
	Wear-leveling [54, 92, 93]	Spreads out writes requests to even out memory cell wear-out.	$\sim$ 50 µs [18]

and storage supports happen in the background, at the memory controller, and are transparent to the processor, therefore, we collectively refer to them as *backend memory operations (BMOs)*. In conventional programs, reads are on the critical path of execution. Hence, these BMOs optimize for read latency, like the counter-mode encryption operation introduced in Section 2.3.2. However, to satisfy the crash consistency guarantees, the write latency can also be on the critical path of PM programs, introducing new research problems on optimizing the write latency. In Chapter 4B, we provide a software-hardware co-design that mitigates the latency of BMOs.

2.4 Security Implications of Persistent Memory Systems

Operations that ensure data security in PM hardware, as introduced in Section 2.3 prevents attackers from reading or tampering data. In this section, we introduce the background on side-channel attacks, which assesses the security aspects of PM from a different angle.

2.4.1 Side-Channel Attacks

Instead of directly exploiting information leakage vulnerabilities in interfaces, side channels observe the behavior of a target system [100], e.g., power consumption, EM radiation, or timing, and deduce secrets from this meta-information.

Cache attacks target the caches of modern processors, with the most techniques being Prime+Probe [101, 102] and Flush+Reload [103]. Both enable a local attacker to observe cache activities of co-located programs via timing differences in memory accesses. Both techniques were used to build fast and stealthy covert channels [103–109], i.e., side channels with a colluding victim exfiltrating data. NetCAT [110] showed that cache timing differences can even be induced and exploited over the network on systems with RDMA or DDIO support. However, Intel recommends disabling RDMA and DDIO in untrusted networks to mitigate the attack. More recently, cache attacks gained substantial attention as building blocks of transient-execution attacks [23, 24, 111–116]. Schwarz et al [117] demonstrated that such attacks can also be exploited remotely.

Previous works reverse-engineered undocumented hardware to assess their attack surface and security relevance. For example, DRAMA exploits DRAM row buffers to establish a covert channel and monitor memory accesses [118], which is enabled by reverse-engineering DRAM addressing functions. Gras et al. [119] exploit the Translation-Lookaside Buffer (TLB) to leak sensitive information such as cryptographic keys, which is enabled by reverse-engineering the TLB internal behavior. These examples show that with co-location and hardware sharing in the cloud, side channels are an immediate threat. We need to find and mitigate these new attacks before they are exploited.

2.4.2 Hardware System inside Optane PM

An Optane PM module consists of several components [120], as shown in Figure 2.5. As a single Optane storage chip has limited performance, these internal components bridge the performance gap. *First*, an Optane DIMM integrates multiple Optane storage chips that can be accessed in parallel for higher bandwidth. *Second*, similar to flash chips in SSDs [121–123], Optane chips also have a limited write endurance [124]. Therefore, the Optane controller performs wear-leveling by changing the mapping between the physical and Optane's internal addresses after a number of accesses. Thus, each access performs a physical-to-internal address translation before accessing the Optane media. *Third*, to hide such translation latency, the DIMM has SRAM and DRAM caches to buffer both data and address translation. *Finally*, the Optane DIMM uses residual capacitors to back up these volatile caching structures to ensure persistence.

As an Optane PM module is a sophisticated system with buffers, caches, and specialized controllers, software developers need to model the performance and runtime behavior to optimize software

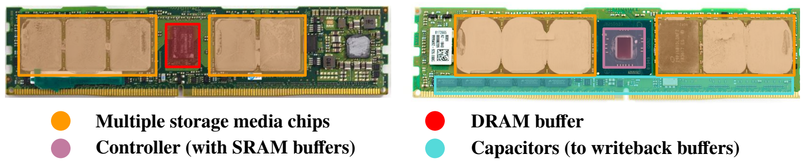

Multiple storage media chips
Controller (with SRAM buffers)
DRAM buffer
Capacitors (to writeback buffers)

Figure 2.5: Components inside an Optane DIMM.

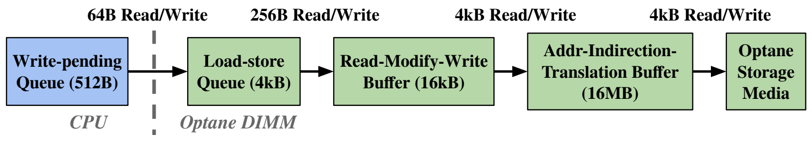


Figure 2.6: Internal memory hierarchy of an Optane DIMM.

systems for Optane. Thus, prior works have characterized the performance metrics of Optane [18–22]. Figure 2.6 illustrates Optane's internal hierarchy, according to their characterizations and Intel's official documentation. On the CPU side, the *Write-Pending Queue (WPQ)* issues 64 B read/write accesses to Optane PM. Correspondingly, on the Optane side, the *Load-Store Queue (LSQ)* accepts the incoming 64 B accesses. After the LSQ, accesses are coalesced into 256 B blocks. These merged accesses then enter a *Read-Modify-Write (RMW)* buffer, which caches 64 entries of 256 B blocks (a total of 64 kB of data), similar to data caches in the CPU. The RMW buffer is also used as a write-back cache, i.e., besides reads, writes also use the RMW. As introduced earlier, the physical address is translated to an Optane-internal address at 4 kB granularity. Thus, if an access misses the RMW buffer, it is translated before accessing the storage media. An *Address-Indirection-Translation (AIT)* buffer maintains a DRAM-based lookup structure to cache 4096 translation entries (covers 16 MB of data in total), much like the CPU's TLB that caches virtual-to-physical address translation.

As Optane has an internal memory system, like CPUs, we study its security properties and whether it facilitates new side-channel attacks in Chapter 5. Existing characterization works [18–22] do not permit such security insights, as security-critical aspects like replacement policy and associativity are unclear.

Chapter 3

Software Support for Persistent Memory Systems

Chapter 3A

Testing Framework for Persistent Memory Programs

3A.1 Introduction

Programming in PM systems for crash consistency is hard and error-prone, as we have introduced in Section 2.2. The two fundamental guarantees required by any crash-consistent software 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` [3]) to efficiently writeback cache lines to memory. We refer to the act of making a cache line persistent (through a writeback or other means [1,40]) as a *persist operation.*

Enforcing ordering is another fundamental necessity for any crash-consistent software. An *ordering guarantee* from the PM system is required for crash-consistent software 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 [37,125] 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 [3,34]. While developing crash-consistent software 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 [35–37, 126], 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 crash-consistent software will *not* result in a runtime ordering that *violates* the required ordering of the *persist* operations. We refer to the bugs that cause PM program to fail recovery as *crash consistency bugs*.

We argue that 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 development of crash-consistent software, they are all specific to certain file systems [127] or user-space libraries [128, 129]. 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 [127], a tool that tests Intel's persistent memory file system (PMFS [130]) can take more than 5 years to test all possible orderings in a trace with around 100k PM operations. In this chapter, 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 crash-consistent 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 uses a strict ordering of writes [3, 34], while recent academic proposals relax this ordering [1, 39, 40]). In this chapter, 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 of the difference in crash-consistent 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 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 software 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 crash-consistent software. Similar to assertions [131, 132] 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 3A.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 3A.5). High-level checkers can automate the process of debugging crash-consistent software 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 [1], 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 [133, 134]. Further, our experiments also reveal that PMTest checkers can help programmers understand the persistency guarantees of PM libraries.

Contributions. The main contributions of this chapter are the follows:

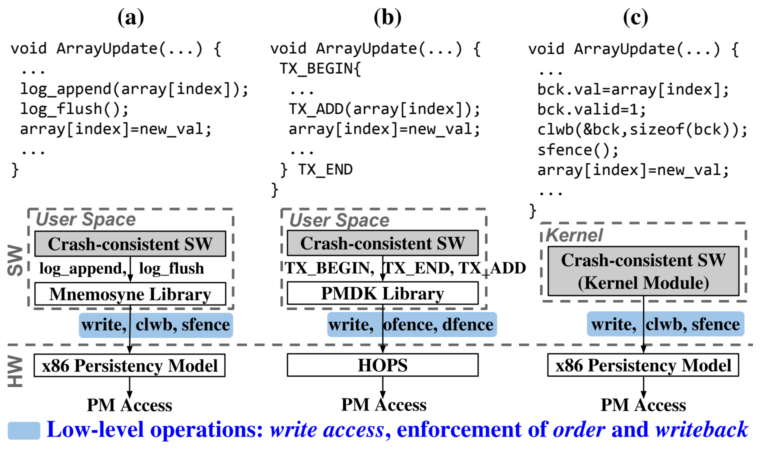
- We design and implement PMTest, a tool to detect crash consistency bugs in PM applications. To our knowledge, PMTest is the first tool that is both flexible and fast.
- PMTest is flexible as it enables the design of specific checkers in the software for different libraries and persistency models. Currently, PMTest supports user-space transaction memory libraries Mnemosyne [36] and PMDK [35] and Intel's kernel-space PM-optimized file system PMFS [130] under the x86 persistency model [3].
- 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 [128].

- PMTest detects 45 synthetic/reproduced bugs and found 3 new bugs in PMDK applications [35] and PMFS [130].

3A.2 Crash-consistency Testing

In Section 2.2, we have demonstrated that it is difficult to implement correct crash-consistent software for PM systems. With low-level hardware primitives, programmers need to carefully manage writes to persistent data. Even with the aid from higher-level libraries, programmers still need to have a good understanding of the failure-recovery requirements of their programs and correctly use the library methods. 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 crash-consistent software and a testing framework must be flexible to support as many as possible. First, there are three types of crash-consistent software systems: *(i)* user-space applications using high-level libraries such as NV-Heaps [37], Mnemosyne [36], and PMDK [35], *(ii)* user-space applications using ISA-specific low-level primitives, such as PM database [135] and key-value stores [136], and *(iii)* kernel-space file systems using low-level primitives, such as PMFS [130] and NOVA [9]. Second, the other variation in crash-consistent software 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 [39] (e.g., the strict persistency model from x86 [3] and the relaxed model proposed by HOPS [1]). 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 crash-consistent software running on various PM systems. Figure 3A.1 shows three possible system stacks and their code examples: (a) a crash-consistent software system developed on top of the Mnemosyne library [36] runs on a system with x86 persistency model, (b) a crash-consistent software system built with the PMDK library [35] runs on the HOPS persistency model that supports more relaxed fences [1], and (c) a persistent kernel module using low-level functions (e.g., PMFS [130]). Ideally, a testing framework should be flexible enough to support all kinds of crash-consistent software systems running on a variety of PM systems.



```
void ArrayUpdate(...) {
 ...
 log_append(array[index]);
 log_flush();
 array[index]=new_val;
 ...
}
```

```
void ArrayUpdate(...) {
 TX_BEGIN{
  ...
  TX_ADD(array[index]);
  array[index]=new_val;
  ...
 } TX_END
}
```

```
void ArrayUpdate(...) {
 ...
 bck.val=array[index];
 bck.valid=1;
 clwb(&bck,sizeof(bck));
 sfence();
 array[index]=new_val;
 ...
}
```

Figure 3A.1: Different PM system stacks and sample codes.

Table 3A.1: Tools for testing crash consistency.

Tool Name	Speed	Flexibility	Target Software	Kernel?
Yat [127]	Low	Low	PMFS [130]	Yes
Pmemcheck [128]	Medium	Low	PMDK [35]	No
PMTest (this work)	High	High	Various types	Yes

Fast. We identify that an efficient crash consistency testing mechanism needs to meet two performance requirements. First, a crash-consistency testing solution 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 [137]. 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 [138–143]. Unfortunately, these tools are designed for block-addressable file systems [138–143], and therefore, cannot be applied to PM-specific crash-consistent software. Second, the tool, Yat [127], that tests Intel's PM-based file system PMFS [130] 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 [127]. Third, there have been faster testing tools developed for specific PM libraries. For example, Pmemcheck [128] (around 20x slowdown) and Persistence Inspector [129] 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 3A.1 summarizes the capabilities of these tools and it is evident that they cannot satisfy both requirements of speed and flexibility.

3A.3 Key Ideas of PMTest

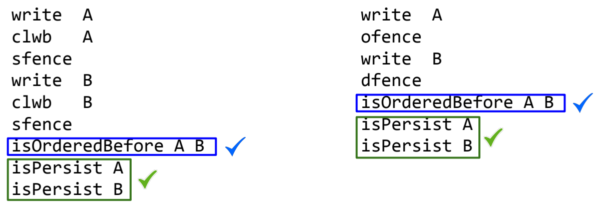
In this work, we propose PMTest, a framework for detecting crash consistency bugs in different crash-consistent software systems running on a variety of PM systems. First, we present our high-level ideas in testing . Then, we discuss how these key ideas are applied to PMTest.

3A.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 as follows:

Flexible. We observe that regardless of the difference in the crash-consistent software systems (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 3A.1 shows that at the lowest level, they all 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 3A.4.4).

Similar to the commonly used assertions [131, 132], 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.2a and 3A.2b 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 low-level checkers in both examples check: *(i)* whether A persists



```
write  A
clwb   A
sfence
write  B
clwb   B
sfence
isOrderedBefore A B ✓
isPersist A
isPersist B ✓
```

(a)

```
write  A
ofence
write  B
dfence
isOrderedBefore A B ✓
isPersist A
isPersist B ✓
```

(b)

Figure 3A.2: Checking mechanism based on the semantics of (a) the x86 persistency model [3] and (b) HOPS [1].

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.

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 [3]. 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 3A.3a shows a trace of PM operations, where

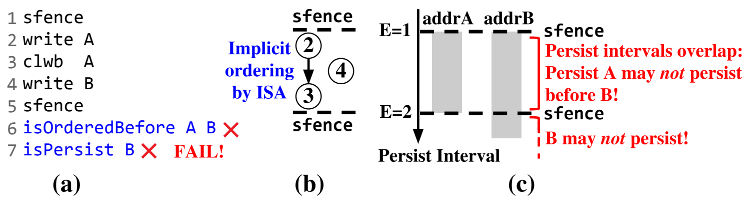


Figure 3A.3: (a) A trace of PM operations. (b) The order between PM operations. (c) The persist interval of writes.

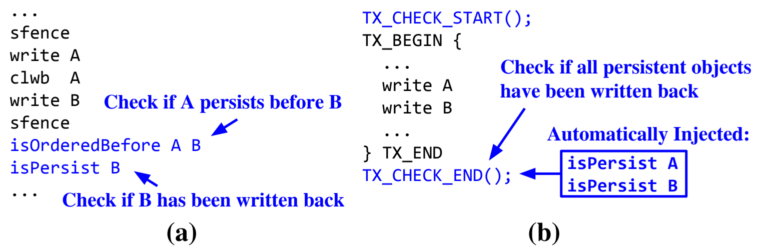
the programmers want to check two issues: if A always persists before B, and if B has been persisted after the last `SFENCE`. Assuming the first `SFENCE` starts the first epoch ($E = 1$), the persist interval for address A is $(1, 2)$, as the write to address A, and the subsequent `CLWB` are both issued 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 ∞.

3A.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 3A.4a shows how to place these checkers to detect crash consistency bugs. Similar to using low-level primitives for programming crash-consistent software, 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 3A.4b 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 3A.5.1). We show that these high-level checkers can effectively detect bugs with minimal programmer's effort in Section 3A.6.3.

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



```
...
sfence
write A
clwb  A
write B
sfence
isOrderedBefore A B
isPersist B
...
```

```
TX_CHECK_START();
TX_BEGIN {
  ...
  write A
  write B
  ...
} TX_END
TX_CHECK_END();
```

Check if A persists before B
Check if B has been written back
(a)
Check if all persistent objects have been written back
Automatically Injected:
isPersist A
isPersist B
(b)

Figure 3A.4: Examples of testing programs using (a) the low-level checkers and (b) checkers for transactions.

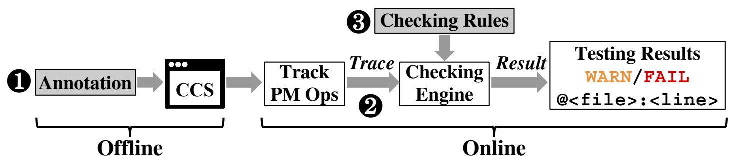


Figure 3A.5: A high-level view of PMTest (shaded components can be customized by programmers).

efficiency, PMTest pipelines the execution of 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 3A.4.4). Decoupling program execution from checker validation provides a marked improvement in performance.

3A.4 Implementation of PMTest

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

3A.4.1 Overview of PMTest

Figure 3A.5 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 test program 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 3A.3). In the *online* step, PMTest executes with the annotated (and compiled) program. During execution time, PMTest tracks PM operations in the application and passes the trace to the checking engine (step ❷, details in Section 3A.4.3). The checking engine tests whether the trace meets the requirements specified by the checkers (step ❸, details in Section 3A.4.4). The checking engine depends on the checking rules to detect the bugs. We discuss the rules for x86 systems in Section 3A.4.4 (already integrated in PMTest) and the rules for HOPS [1] in Section 3A.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.

3A.4.2 PMTest Interface

PMTest incorporates a flexible software interface that is C and C++ compatible. Table 3A.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 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 to 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.

3A.4.3 Operation Tracking

A trace in PMTest consists of the PM operations executed by crash-consistent software and the checkers placed by programmers. Each PM operation in the trace has associated metadata that

Table 3A.2: Summary of PMTest functions.

	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
Communication	`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

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, we extend the existing tracking mechanism in the PM benchmark suite, WHISPER [1], which converts all PM operations into macros for benchmarking purposes. We extend their tracking method by adding PMTest tracking functions to generate the aforementioned metadata for PM operations (e.g., writes, `CLWB` and `SFENCE` in x86). For other crash-consistent software systems, it is possible to either integrate a WHISPER-like tracking mechanism or to use a toolchain (e.g., through an LLVM [144] pass) that injects a tracking function for each PM operation.

3A.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 [145], 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_interval`s and `flush_interval`s within the address range and sets the `persist_interval`s 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 3A.5.1).

• **`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:

```
1 write(0x10,64)
2 clwb(0x10,64)
3 sfence()
4 write(0x50,64)
5 isPersist(0x50,64)
6 isOrderedBefore
    (0x10,64,0x50,64)
```

(a) Trace

		0x10~0x4f		0x50~0x8f	
Op#	**T**	**PI**	**FI**	**PI**	**FI**
❶	0	(0,∞)			
❷	0	(0,∞)	(0,∞)		
❸	1	(0,1)	(0,1)		
❹	1	(0,1)	(0,1)	(1,∞)	

❻ 0x10 will persist before 0x50

❺ Not persistent

(b) Update steps

Figure 3A.6: An example of checking a trace.

• **`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 the address range $[addrA, addrA + sizeA)$ 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 $[addrA, addrA + sizeA)$.

Example. Figure 3A.6a shows a sample trace, and Figure 3A.6b 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 $(0, \infty)$.

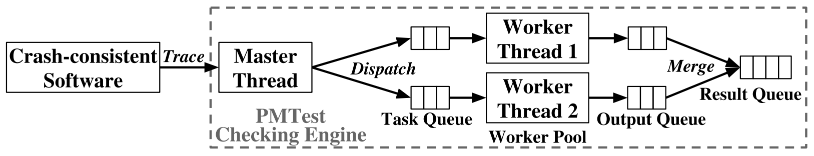
Line 3: The `SFENCE` first increments the timestamp T. Then, it updates the FI of its preceding `CLWB` to $(0, 1)$, indicating this writeback will take effect before line 3. It also updates the PI for 0x10 to $(0, 1)$, indicating that this write has persisted.

Line 4: The `write` updates the PI for address 0x50 to $(1, \infty)$.

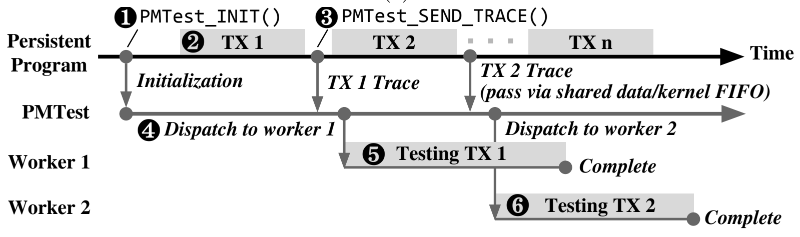
Line 5: The `isPersist()` checker examines the PI of 0x50. As $(0, \infty)$ does not end before the current T, this checker reports a `FAIL` output as indicated by the red arrow.

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 3A.7a. The master thread dispatches the traces passed from the test program (details about communication between the program and PMTest in Section 3A.4.5)



(a)



(b)

Figure 3A.7: (a) The master and worker threads and (b) the workflow of PMTest.

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 3A.7b 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).

3A.4.5 System Integration

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

User-space Crash-consistent Software. Figure 3A.8a shows the system stack of testing a user-space crash-consistent software. The user-space crash-consistent software runs in the same process as the PMTest checking engine. To efficiently pass traces from the test program to the checking engine, we use a thread-safe, concurrent queue, where the test program 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.

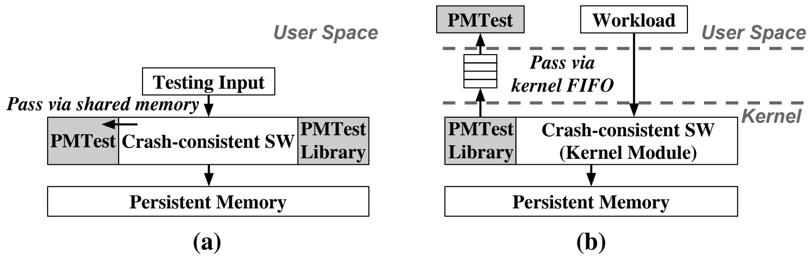


Figure 3A.8: System integration of PMTest for (a) user-space programs and (b) kernel modules.

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 3A.8b 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 [146, 147] (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 [147] 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.

3A.5 Flexibility of PMTest

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

3A.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 [35], and then present other checkers that detects performance bugs.

Library-Specific Checkers

Library-specific, high-level checkers can automate the debugging for crash-consistent software 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 2.1b). A 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 objects logged by `TX_ADD()` (or functions with similar functionality), together with other operations. The checking engine maintains another interval tree, *log tree*, that keeps tracks of 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`.

Performance Checkers

We provide the implementation of two checkers for detecting 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`.

3A.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. Implementing 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 [1, 40, 148]. The hands-off persistence system (HOPS) [1] 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 3A.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_interval`s 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`.

Table 3A.3: System Configuration.

Server	HP ProLiant DL360 Gen10
Processor	Intel Skylake, 2.1GHz, 8 cores, 16 threads, 11 MB L3 [150]
Memory	Volatile: 64 GB DDR4, 2666 MHz Non-Volatile: 64 GB Battery-backed NVDIMM
OS	Ubuntu 14.04, Linux kernel 4.4.135
Compiler	gcc/g++ 4.8.4, O3 optimization

- **`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 $[addrB, addrB + sizeB)$.

3A.6 Evaluation

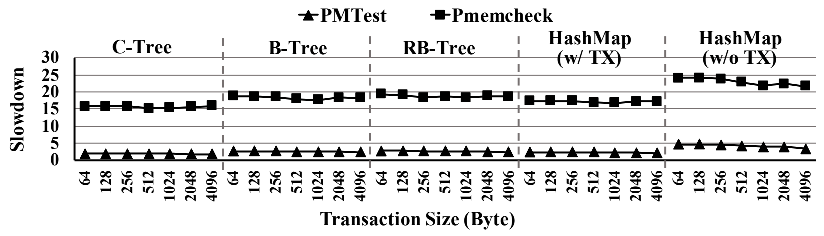
In this section, we evaluate the performance and bug detection capability of PMTest.

3A.6.1 Methodology

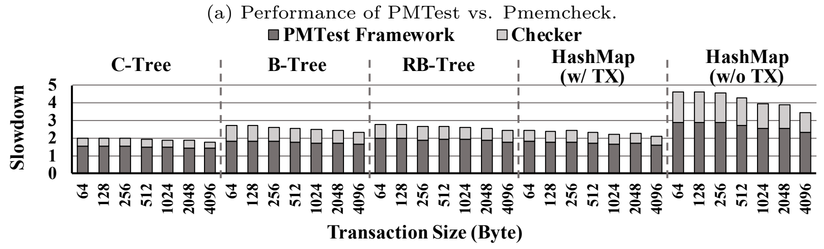
To evaluate the performance and bug detection of PMTest, we use a real system as shown in Table 3A.3. We use a set of battery-backed NVDIMMs as the PM and map them to the system following the method in [149]. We use programs from the WHISPER benchmark suite [1] to evaluate both performance and bug detection. PMTest performs testing using one worker thread unless explicitly indicated. The execution times shown in this section are the average of ten runs.

3A.6.2 Performance Evaluation

Microbenchmark. We evaluate PMTest using five PMDK-based single-threaded microbenchmarks. We test each program with 100K insertions (each insertion is a transaction). Figure 3A.9a 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



(a) Performance of PMTest vs. Pmemcheck.



(b) Overhead breakdown of PMTest.

Figure 3A.9: Performance of testing microbenches.

Table 3A.4: Real workloads from WHISPER benchmark suite [1] (YCSB from [2]).

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

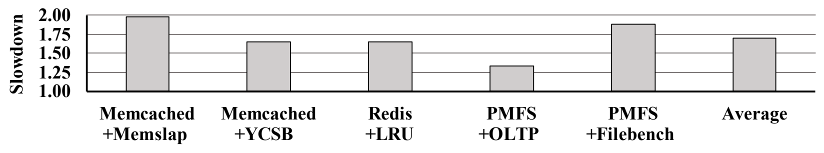
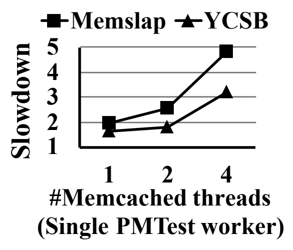


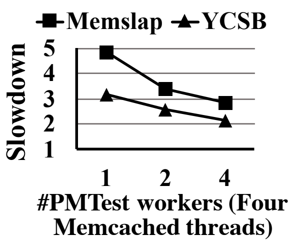
Figure 3A.10: Performance of testing real workloads.

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 3A.9b, 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.

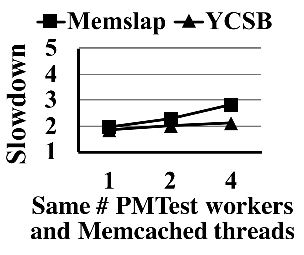
Real Workloads. We evaluate three real workloads shown in Table 3A.4, where each of them has its own load-generating client(s). We place the checkers to test whether all updates in the transactions (as specified by WHISPER) are persistent in PMFS [130] and Mnemosyne [36], and use our transaction checkers in Redis. Figure 3A.10 shows the performance of these workloads running



(a)



(b)



(c)

Figure 3A.11: Execution time of Memcached with PMTest.

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.

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 3A.11a 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 3A.11b. 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 3A.11c shows the slowdown slightly increase as both threads increase due to the inter-thread communication overhead. We conclude that PMTest can effectively reduce the testing time when testing PM-operation intensive programs.

3A.6.3 Bug Detection Evaluation

To validate the bug detection capability of PMTest, we first systematically create random synthetic bugs in PMDK workloads [35]. Table 3A.5 lists the synthetic bugs we have validated (total 42)[1]. 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 2.6 k LOC). PMTest reported all the synthetic bugs we

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

Table 3A.5: Summary of synthetic bugs for PMTest validation.

	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 (High-level checkers)
	Completion	Incomplete transactions due to improper termination	7	
	Performance	Log the same persistent object more than once	4	

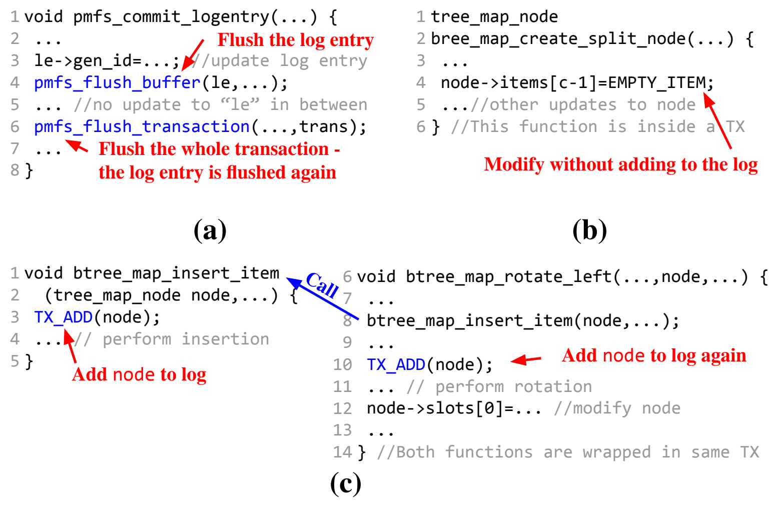
Table 3A.6: Summary of the known bugs in the commit history and the new bugs detected by PMTest.

	File	Line	Description
Known	`xips.c` [151]	207, 262	Flush the same persistent buffer twice
	`files.c` [152]	232	Flush an unmapped buffer
	`rbtree_map.c` [153]	379	Modify a tree node without logging it
New	`journal.c` [154]	632	Flush redundant data when committing
	`btree_map.c` [155]	201	Modify a tree node without logging it
		367	Log the same object twice

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 3A.6). Figure 3A.12 demonstrates the new bugs we have found using PMTest. We simplify the code for readability.

Bug 1 (performance): Figure 3A.12a 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 3A.12b shows a snippet of code from `btree_map.c` in PMDK. This function modifies a `node` 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



```
1 void pmfs_commit_logentry(...) {
2 ...
3 le->gen_id=...; //update log entry
4 pmfs_flush_buffer(le,...);
5 ... //no update to "le" in between
6 pmfs_flush_transaction(...,trans);
7 ...
8 }
```

```
1 tree_map_node
2 bree_map_create_split_node(...) {
3 ...
4 node->items[c-1]=EMPTY_ITEM;
5 ...//other updates to node
6 } //This function is inside a TX
```

```
1 void btree_map_insert_item
2   (tree_map_node node,...) {
3 TX_ADD(node);
4 ...// perform insertion
5 }
```

```
6 void btree_map_rotate_left(...,node,...) {
7  ...
8  btree_map_insert_item(node,...);
9  ...
10 TX_ADD(node);
11 ... // perform rotation
12 node->slots[0]=... //modify node
13 ...
14 } //Both functions are wrapped in same TX
```

Flush the log entry
Flush the whole transaction -
the log entry is flushed again
(a)
Modify without adding to the log
(b)
Call
Add node to log
Add node to log again
(c)

Figure 3A.12: New bugs found in (a) PMFS, and (b, c) PMDK applications.

from Intel can be found at [133].

Bug 3 (performance): Figure 3A.12c 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 [134].

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.

3A.7 Discussion

In this section, we discuss the opportunities and potential issues with using PMTest, and the future works in testing crash-consistent software.

3A.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.

3A.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 2.1). 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 crash-consistent software built with high-level libraries.

3A.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.

3A.7.4 Future Work

In this section, we describe the future directions.

Dynamic v.s. Static Testing. 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 Crash-consistent Software. 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 [1]. We leave debugging crash consistency issues due to improper thread synchronization as a future work.

Chapter 3B

Testing for Persistent Memory Programs across System Failures

3B.1 Introduction

In Chapter 3A, we have introduced our runtime testing framework, PMTest [25] that detects crash consistency bugs by checking the persistence and ordering properties of PM operations at runtime. Required by the crash consistency guarantee, that is a program returns to a consistent state and resumes the execution after a failure, a testing tool is expected to detect inconsistencies during the *entire* procedure of execution, recovery, and resumption. Therefore, testing the program during normal execution only covers part of the testing scope. In this chapter, we identify that a crash-consistent program must ensure a correct interaction between the execution stage *before* and *after* the failure. Therefore, a program *first* needs to correctly implement certain crash consistency mechanisms (e.g., undo/redo logging [35, 36, 125, 156–159], checkpointing [47, 160], or shadow paging [161, 162]) to ensure data consistency *before* failure. And *second*, after failure, the associated recovery procedure must properly restore PM to a consistent state. We refer to the stages before and after the failure as the *pre-failure* and *post-failure* stages. The pre- and post-failure stages are required to work collaboratively to guarantee crash consistency. If the interaction between the two stages is incorrect, the program might not recover to a consistent state. In the previous undo logging example, even if the program correctly maintains undo logs during the pre-failure stage, the post-failure execution might still read inconsistent data if the recovery procedure does not correctly roll back incomplete updates according to the undo logs. Hence, both the pre- and post-failure execution stages are critical to the crash consistency guarantee. In this work, we seek to test the crash consistency guarantee holistically, considering both the pre- and post-failure execution stages.

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

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

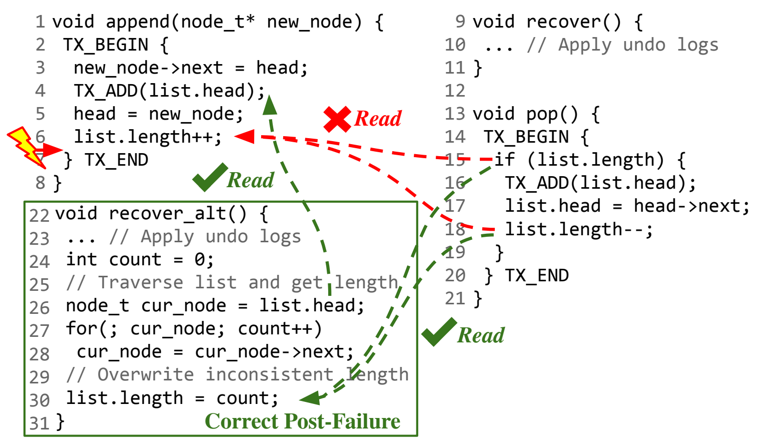
Even in the absence of cross-failure races, the program can still be semantically incorrect and cause inconsistencies across the failure. For example, under the checkpointing-based recovery mechanism, the post-failure execution should read only from data in the most recent committed checkpoint. Data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

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

The contributions of this chapter are the following:

- This work shows that the crash consistency guarantee relies on the correct interaction between the pre- and post-failure stage of a PM program.
- We categorize the incorrect cross-failure interactions into two classes: (1) cross-failure race, where the post-failure execution reads from non-persisted data, and (2) cross-failure semantic bug, where the post-failure execution reads from semantically inconsistent data.
- Based on the categorization and definition, we implement XFDetector[1], a tool that automatically injects failures into programs, and detects cross-failure bugs by replaying traces of the pre- and post-failure stages.
- XFDetector has detected four new bugs in three pieces of PM software: one of PMDK's examples, a PM-optimized Redis [14] database, and a PMDK library function [35].

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



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

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

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

Figure 3B.1: An example of an inconsistency in program's post-failure execution.

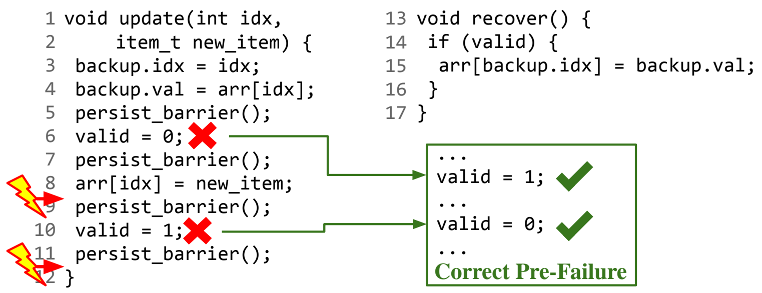
3B.2 Background and Motivation

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

3B.2.1 Need for An End-to-End, Cross-Failure Testing

In Chapter 3A, we have described our testing framework, PMTest [25], that performs a runtime testing to check whether the normal execution of a PM program meets the requirements for persistence and ordering. However, only testing the normal execution stage is insufficient as crash consistency has two fundamental requirements: (1) the program needs to correctly follow crash consistency mechanism to ensure data consistency *before* a failure happens, and (2) the recovery code needs to correctly restore the PM status back to a consistent state after a failure and resume the previously preempted execution. For simplicity, we refer to the phase before failure as the *pre-failure* stage, and after failure as the *post-failure* stage. Next, we will show two examples that fail to meet these requirements.

Example 1: Inconsistency in the post-failure execution. Figure 3B.1 shows a snippet of code that appends a `new_node` to a persistent linked list. To guarantee crash consistency, it wraps the updates in a transaction (indicated by `TX_BEGIN` and `TX_END`). Within the transaction, it adds the current PM object to an undo log with a `TX_ADD()` function (line 4), such that if a failure happens in the middle of the transaction, the recovery program can roll back the logs and restore to a consistent state. However, the program does not add `length` to the undo log. As a result, if a failure happens between line 6 and 7, it is unknown if the `length` of the linked list has been persisted. Whether or



```
void update(int idx,
    item_t new_item) {
 backup.idx = idx;
 backup.val = arr[idx];
 persist_barrier();
 valid = 0;
 persist_barrier();
 arr[idx] = new_item;
 persist_barrier();
 valid = 1;
 persist_barrier();
}
```

```
void recover() {
 if (valid) {
  arr[backup.idx] = backup.val;
 }
}
```

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

Figure 3B.2: An example of an inconsistency in program's pre-failure execution.

not this inconsistent `length` can lead to a bug depends on the post-failure execution.

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

To recover the linked list to a consistent state without requiring the logged `length`, `recover_alt()` traverses the linked list and gets the number of nodes (line 26-28) after applying the undo logs. Then, it *overwrites* the `length` with the correct value (line 30), making the variable `length` consistent. During traversal, the program reads from the consistent value of `head` as it has been backed up by the transaction (indicated by the green arrow). And, after executing the `recover_alt()` function, the function `pop()` also accesses a consistent version of `length` that has been overwritten during the recovery (indicated by the green arrows). Note that the update to `length` at line 30 does not need to be covered by a transaction because its value always gets reset during recovery. Compared to adding `length` to the transaction during the pre-failure stage, this fix is more efficient as the recovery procedure only happens once for each failure. Thus, we refer to this example as an inconsistency in the post-failure stage. However, even with a correct implementation of `recover_alt()`, existing works in crash consistency testing [25, 128] can report a false positive as they only check the pre-failure stage.

Example 2: Inconsistency in the pre-failure execution. Figure 3B.2 shows a snippet of

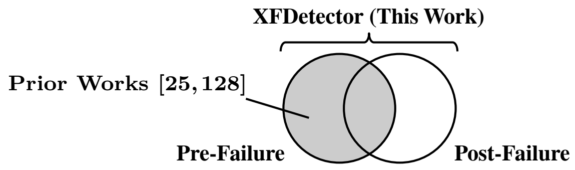


Figure 3B.3: Causes of inconsistency after system failure.

code that updates a location `idx` in a persistent array (`arr`). To guarantee crash consistency, the `update()` function first backs up the old data and the updated index (line 3-4). Then, it issues a `persist_barrier()`to writeback the backup and sets a `valid` bit (line 6). After writing back `valid` with another `persist_barrier()`, it performs the in-place update to the array (line 8). And finally, it persists the updates and resets the `valid` bit (line 9-11). Even though this example places a `persist_barrier()`at the correct places, the pre-failure code is still semantically incorrect as `valid` is set to wrong values (corrections are shown in the green box). As a result, the recovery function always performs the wrong operation: If a failure happens before the in-place update has been written back (line 8), the recovery program observes a `valid = 0` and does not roll back the potentially *non-persisted* update. And, if a failure happens after the `update()` function (line 12) has completed, the recovery program rolls back with the *stale* data that is *semantically inconsistent*. Although the bug fix can apply to both pre- and post-failure stages, the more appropriate way is to change the values in the pre-failure stage as the variable `valid` refers to the validity of the backup. For this reason, we refer to this bug as an inconsistency in pre-failure stage. As the consequence of this bug appears after the failure, prior works [25, 128] cannot detect the bug either.

From these two examples, we conclude that it is hard to guarantee crash consistency, not only because PM programming requires a good knowledge of PM low-level instructions and libraries, but also because the pre- and post-failure stages in the program need to work seamlessly. The inability to implement a correct crash consistency mechanism for the pre-failure execution leaves inconsistent data in PM, making it impossible for post-failure execution to restore PM to a consistent state. On the other hand, an incorrect recovery and resumption execution is unable to consistently restore PM. Figure 3B.3 summarizes these two buggy scenarios where the inconsistencies can be due to the pre-failure and/or post-failure execution. Prior works [25, 128] have provided testing tools to detect crash consistency bugs in the pre-failure stage (the shaded area). However, without performing an *end-to-end* test with both stages involved, it is impossible to cover all buggy scenarios.

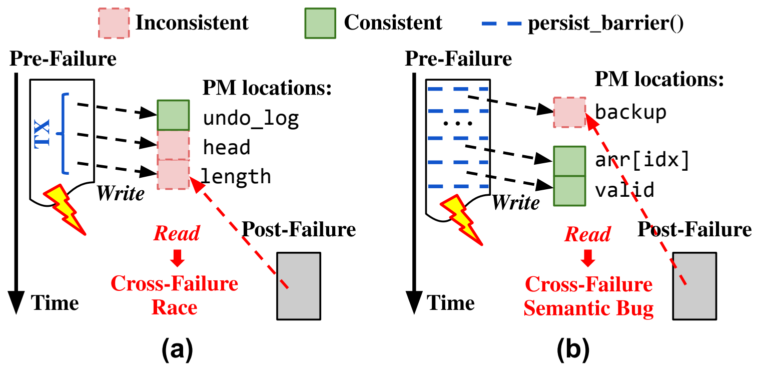


Figure 3B.4: Cross-failure bugs from the example of (a) Figure 3B.1 and (b) Figure 3B.2.

3B.2.2 Causes of Inconsistency

We categorize the incorrect interactions between the pre- and post-failure execution into two classes. The first class of bugs happens when the post-failure execution reads from data that may have not been persisted before the failure. Prior works have suggested that there is a similarity between multithreaded programs and the recovery in certain crash-consistent programs. Lucia et al. model intermittent computing in energy-harvesting devices as concurrency [163, 164]. Chakrabarti et al. make an analogy between races in multithreaded programs and buggy scenarios in their failure-atomic programming model [159]. We further generalize this interaction in PM programs — the execution *before* and *after* a failure can be modeled as a *writer* and a *reader* from two concurrent threads. In the conventional data race, a race happens when at least one of the concurrent accesses to the same memory location is a write [165]. In PM programs, although the pre- and post-failure execution cannot perform real concurrent accesses as they happen in different times, this contentious interaction is still similar to a data race as the value returned by the read after a failure is indeterminate, depending on when the failure happens. Therefore, such a read from a potentially non-persisted location may cause undefined behaviors afterward. We refer to reading data that is not guaranteed to be persisted in the post-failure stage as a *cross-failure race*. Figure 3B.4a illustrates the cross-failure race (indicated by the red arrow) between the pre- and post-failure stages in the example of Figure 3B.1. Due to a post-failure bug, the program fails to overwrite the potentially non-persisted `length` modified by the pre-failure "writer", and thus, the post-failure "reader" can access a non-deterministic value.

The second class of bugs happens when the post-failure program reads semantically inconsistent data. Different from the cross-failure race, where the persistence of data is unknown, this type of cross-failure interaction is *always* incorrect as the actual implementation violates the semantics

Table 3B.1: Data consistency requirements in different crash consistency mechanisms.

Mechanism	**Description**	**Data Consistency**
Undo logging [35, 37, 166–168]	Keeps a backup of the old data before performing the in-place update. If a failure happens during the transaction, the recovery mechanism reverts the update with the backup.	If the transaction has been committed, the updated data is consistent. Otherwise, the log is consistent.
Redo logging [36, 169, 170]	Performs updates to the log instead of updating in place. If a failure happens during the transaction, the recovery mechanism discards the incomplete redo log.	If the redo log has not been committed, the existing data is consistent. Otherwise, the committed log is consistent.
Checkpointing [47, 160, 171]	Creates a checkpoint (i.e., snapshot) of persistent data periodically. After a failure, the recovery mechanism reverts to the last committed checkpoint.	Data in the latest committed checkpoint is consistent.
Shadow paging [126, 162, 172]	Performs copy-on-write such that data under modification has a separate copy. Once all updates to the shadow object are completed, the mechanism swaps the original data with the shadow object (e.g., by atomically updating a persistent pointer).	If the shadow object has been committed, data in the shadow object is consistent. Otherwise, the old data is consistent.
Operational logging [173, 174]	Logs operations instead of data. If a failure happens during the operation, the recovery mechanism re-executes the logged operation to overwrite the incomplete operation.	Logged operations are consistent.
Checksum-based recovery [36, 141, 175]	Determines the consistency status of the modified data using checksums. If a failure happens, the recovery program first reads the data in place and then uses its checksum to determine the consistency.	Data protected by the corresponding checksum is consistent.

of the intended crash consistency mechanism. Therefore, we name the act of reading data that is semantically inconsistent during the post-failure stage as a *cross-failure semantic bug.* Figure 3B.4b shows the cross-failure semantic bug in the example of Figure 3B.2. Due to the pre-failure bug that incorrectly sets the values of `valid`, the post-failure recovery program reads from a semantically inconsistent `backup`. Because the `valid` bit is incorrectly set by the program implementation, the status after the recovery is always incorrect.

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

3B.3 Cross-Failure Bugs

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

3B.3.1 Cross-Failure Race

Definition: *The post-failure execution reads from data modified by the pre-failure execution that is not guaranteed to be persisted before the failure.*

The first type of cross-failure race covers the most general case of inconsistent data on PM — it happens when writes to PM are not guaranteed to be written back before a failure. As data may not be persistent, the post-failure execution can read incompletely updated data, leading to inconsistencies after failure. Reading the variable `length` during post-failure recovery in Figure 3B.1 is a typical example of a cross-failure race as `length` is not guaranteed to be persisted before failure. Its unknown persistence status can lead to uncertainties during the post-failure stage. To formalize the cross-failure race, we first define the following notations:

• W_x: A write to the PM location x.

• R_x: A read from the PM location x.

• M_x: A read/write from/to the PM location x.

• F: A failure point that preempts execution.

We then define the following ordering notations:

• $M_x <_{hb} F$: M_x happens before the failure F.

• $W_x \leq_p W_y$: W_y may not persist before W_x is persisted.

• $W_x \leq_p F$: W_x has been persisted before the failure F.

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

$$W_x <_{hb} F \bigwedge F <_{hb} R_x \bigwedge \neg \left(W_x \leq_p F\right). \tag{3B.1}$$

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

Benign Cross-Failure Race: *A program intentionally reads from potentially non-persisted data modified by the pre-failure execution, without causing inconsistencies.*

Cross-failure races can cause inconsistencies, however, *not all* cross-failure races lead to inconsistencies. Instead, it is sometimes necessary to read potentially non-persisted data to correctly recover from a failure, analogous to the inherent data races on synchronization primitives. We refer to such intentional reads to inconsistent data as the *benign cross-failure race.* For example, reading the valid bit of undo logs during the post-failure recovery is regarded as a benign race, as the valid bit enables the recovery program to determine which version is consistent. The checksum-based recovery mechanism (last row in Table 3B.1) is another example of the benign cross-failure race, as the post-failure recovery needs to read potentially non-persisted data and its associated checksum to verify data consistency. In these scenarios, a write to such location inherently races with the post-failure read, while the outcome is always well defined and thus, does not cause any inconsistency. Benign cross-failure races are typically used to determine the consistency status of other PM objects.

3B.3.2 Cross-Failure Semantic Bug

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

The second type of cross-failure bug covers inconsistencies defined by the program semantics. PM programs typically follow certain crash consistency mechanisms. Even if a PM location is persisted before failure, it can still be semantically inconsistent if it violates the corresponding data consistency requirements. Table 3B.1 lists the data consistency requirements of common crash consistency mechanisms. Among these different mechanisms, we identify that most crash consistency mechanisms keep two versions of data: a consistent version for recovery and another for the current update. The version that is regarded as consistent by the crash consistency mechanism can be safely read during the post-failure execution. Whereas, the inconsistent version should be discarded or overwritten. These mechanisms typically use a commit variable to indicate whether a set of PM addresses belongs to a consistent version. Data in a set of PM addresses are regarded as consistent only if they were updated between the last two updates to the associated commit variable. For example, in the undo logging mechanism, the program first logs the original data and sets the commit variable (a valid bit) of the log. Then, it performs the in-place update and unsets the commit variable. If a failure

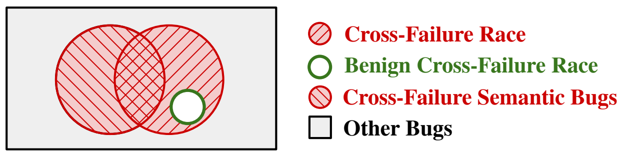


Figure 3B.5: Two classes of cross-failure bugs.

happens after the last update to the commit variable, then the in-place update is the consistent version, as it was modified between the last two updates to the commit variable.

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

• C_{x_i}: The i-th write to the PM address x that alters the consistency status of other PM addresses. We refer to the write as a *commit write* and variable on x as a *commit variable*.

• S_x: A set of PM addresses, i.e., $\{m_1 ... m_n\}$, associated with the commit variable on x.

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

$$S_x \cap S_y = \varnothing. \tag{3B.2}$$

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

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

3B.3.3 Summary

The Venn diagram in Figure 3B.5 summarizes the two classes of cross-failure bugs. The first class of cross-failure bug is the cross-failure race that reads data not guaranteed to be persisted before a failure, unless it is an intended benign cross-failure race. The second class is the cross-failure semantic bug that reads semantically inconsistent data. As the focus of this work is to detect crash consistency bugs due to cross-failure interactions, we do not consider other types of bugs. Next, we describe our key ideas for detection based on the definition of these cross-failure bugs.

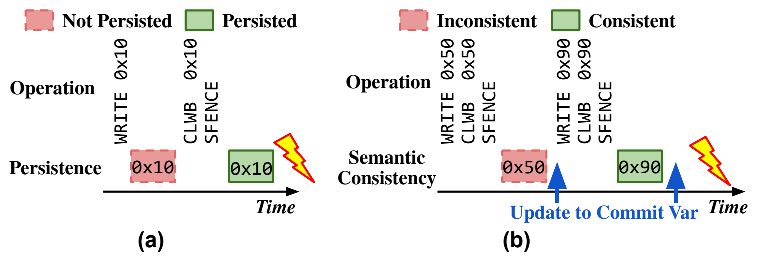


Figure 3B.6: Examples of detecting (a) a cross-failure race and (b) a cross-failure semantic bug based on the data consistency status of PM locations.

3B.4 Key Ideas of XFDetector

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

3B.4.1 Data Consistency

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

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

Figure 3B.6a shows an example of detecting a cross-failure race based on the persistence of data. A PM location, `0x10`, first gets modified and then the persistence status becomes *not persisted* as this update is not guaranteed to be written back. Then, a sequence of `CLWB` and `SFENCE` writes back this location and thus, changing the status to *persisted*. Figure 3B.6b shows another example that detects cross-failure semantic bugs by determining the data consistency status according to the updates to the *commit variable* (indicated by the blue arrows). There are two updates to the locations `0x50` and `0x90` that have been persisted before the failure. However, being persistent does not mean the locations are consistent. As the location `0x90` is last modified between the last two updates to the commit variable, it is regarded as semantically consistent, while the other location `0x50` is not.

3B.4.2 Failure Injection

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

Solution. We observe that updates to PM are not guaranteed to be persisted until explicitly written back (e.g., using a `persist_barrier()`). We refer to a point in the program that explicitly writes back data to PM before any future PM operations as an *ordering point*. As such, persistent data can only transition from an inconsistent state to a consistent state after an ordering point. Therefore, it is only necessary to check the consistency status immediately *before* each ordering point. Based on this observation, XFDetector only injects failure points before each ordering point[2]. The ordering points that XFDetector concerns about include both low-level operations (e.g., `SFENCE`) and high-level functions that enforce writeback (e.g., `TX_ADD()` in PMDK [35]).

3B.5 Implementation of XFDetector

3B.5.1 An Overview of XFDetector

Figure 3B.7 shows an overview of XFDetector's detection procedure that consists of three steps: (1) an *offline* step that requires annotation of the region-of-interest (RoI) in both the pre- and post-

[2]Checksum-based mechanism is an exception that data consistency relies on the verification of the checksum. We briefly discuss how to inject additional failure points for this mechanism in Section 3B.5.5.

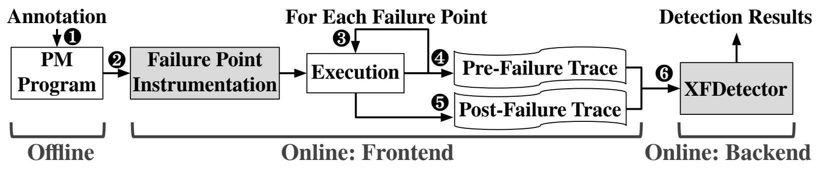


Figure 3B.7: An overview of XFDetector.

failure stages, (2) an *online frontend* that injects failure points and generates traces, and (3) an *online backend* that detects and reports cross-failure bugs based on the traces. The following is an overview of the detection procedure: First, the programmer annotates the source code and compiles it with XFDetector library (step ❶). Second, XFDetector automatically instruments the program with failure points before its execution begins (step ❷). During execution, it follows a procedure of execute pre-failure stage – suspend at the failure point – execute the corresponding post-failure stage, until it completes or reaches the termination point (step ❸). During execution, it generates both the pre-failure (step ❹) and post-failure traces (step ❺). Finally, as the frontend is tracing, the backend performs detection and reports the detection results (step ❻).

3B.5.2 Software Interface

XFDetector provides a C/C++-compatible interface as listed in Table 3B.2. XFDetector has two types of functions. The first type controls the detection procedure and allows programmers to select the region-of-interest (RoI) for detection. The second type is used for annotating the source code to support detection. For trusted code (e.g., implementation of library functions), programmers can choose to skip the injection of failure points and bug detection. Programmers can also add additional failure points on demand. To expose crash consistency semantics in programs directly built on low-level primitives, XFDetector allows programmers to register the commit variable and its associated PM objects. By default, if there is only one commit variable and no object is specified, it covers all PM locations. During the execution of XFDetector, reads from the selected commit variables are marked as benign cross-failure races, without being reported as bugs. Both types of functions take two arguments, `condition` and `stage`, which allow programmers to manage when the function takes effect. It is worth pointing out that programmers only need to use the functions to select the region for detection, without any need for additional annotation when testing programs that are built on top of PM libraries. The functions for annotation are needed only when testing programs that directly use low-level primitives or the implementation of PM libraries.

Table 3B.2: XFDetector software interface.

	Function	Description

Control	RoIBegin(condition, stage)
	RoIEnd(condition, stage)
		Mark a region for XFDetector detection

	completeDetection(condition, stage)	Terminate detection

Detection Annotation	skipFailureBegin(condition)
	skipFailureEnd(condition)
		Mark a region that skips failure points

	addFailurePoint(condition)	Add additional failures

	skipDetectionBegin(condition, stage)
	skipDetectionEnd(condition, stage)
		Mark a region that skips detection

	addCommitVar(variable)
	addCommitRange(variable, addr, size)
		Mark a commit variable and associated address

3B.5.3 Tracing Mechanism

XFDetector's tracing mechanism generates a trace of low-level instructions, including PM reads and writes, fences, and writeback operations. XFDetector leverages PMDK's address derandomization option to map PM locations to a predefined virtual address range to distinguish PM operations, and keeps the virtual address of each PM object the same across different executions to simplify the detection process (by setting the `PMEM_MMAP_HINT` environment variable [176]). Due to the high performance overhead from tracing at such fine granularity, XFDetector optimizes the tracing procedure for programs built on PMDK [35] by skipping the trace of internal implementation but only maintaining a trace of library function calls, such as PM transactions and allocations. This way, the user code is traced at instruction granularity and internal library code is traced at function granularity. In the trace entry, XFDetector keeps track of the operation's instruction pointer, and the source/destination addresses and their sizes. The instruction pointer is used for backtracing the bug, and the address and size differentiates PM objects.

3B.5.4 Detection Procedure

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

Frontend. We implement the frontend based on Intel's Pin [177] to perform tracing and failure injection. In order to inject failures for testing, before executing the program, the frontend first locates all ordering points in the binary and then instruments the binary with failure handlers before each ordering point (as described in Section 3B.4). After instrumentation, the frontend performs tracing and failure injection during execution (as shown in Figure 3B.8a). In the *pre-failure stage*,

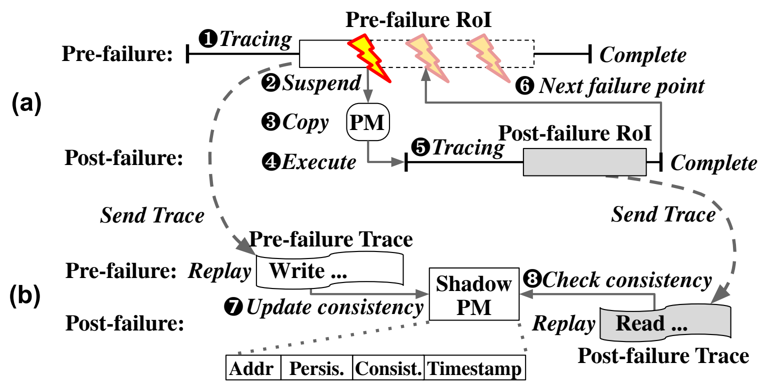


Figure 3B.8: XFDetector's (a) frontend and (b) backend.

XFDetector collects a trace of PM writes and library functions (step ❶). When encountering a failure point (i.e., failure handler) within the RoI, XFDetector suspends the program (step ❷), makes a copy of the current PM image (a pool file on PM) (step ❸)[3], and spawns its post-failure execution (step ❹). Then, in the *post-failure stage*, XFDetector generates another trace (step ❺) until it reaches the annotated termination point (or naturally terminates). Then it continues the pre-failure execution and moves on to the next failure point (step ❻). During the remaining pre-failure execution, XFDetector incrementally traces new operations instead of starting over from the beginning for better performance. While tracing, the frontend sends the already completed trace to the backend (through a pre-failure trace FIFO and a post-failure trace FIFO) for detection. This way, the detection procedure can overlap with tracing. Next, we describe the backend detection mechanism.

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

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

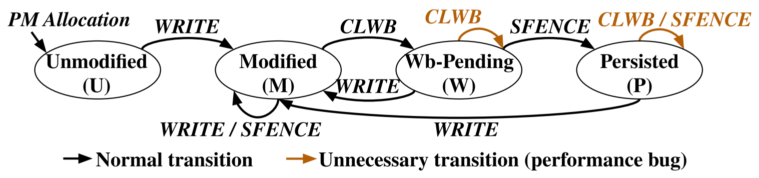


Figure 3B.9: Transitions of the persistence state.

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

Pre-failure Trace: XFDetector's backend replays the pre-failure trace by updating the shadow PM for each write and each library function that modifies PM (step ❼). XFDetector updates both the *persistence state* and *consistency state* according to the operations in the trace. For the persistence state of a PM location, XFDetector follows the finite-state machine in Figure 3B.9. In brief, a `WRITE` changes the state to *modified*, a `CLWB` changes the modified state to *writeback-pending*, and finally, an `SFENCE` changes the state to *persisted*[4]. For consistency state, this work supports common crash consistency mechanisms that use commit variables and those that are built on PMDK transactional functions[5]. Figure 3B.10 shows the consistency state transition of a PM location m according to the updates to its associated commit variable x (C_x refers to a write to the commit variable), where the location m can be *inconsistent* in two ways: being uncommitted or stale. In brief, an uncommitted location becomes consistent after an update to the commit variable, and a stale location first gets updated and then becomes consistent after being committed. XFDetector handles PMDK transactional functions in a similar way to PMTest [25], where objects that have been added to the transaction are regarded as consistent. During the update of the shadow PM, XFDetector also reports performance bugs that use unnecessary PM operations (e.g., redundant writebacks as indicated by the yellow edges in Figure 3B.9), and unnecessary library functions (e.g., duplicated `TX_ADD()` functions for the same PM object).

Post-failure Trace: XFDetector then replays the corresponding post-failure trace (step ❽). Different from processing the pre-failure trace, writes in post-failure change the consistency status to *consistent* as they *overwrite* the old data. Inconsistencies introduced by these writes will be tested later when this code region runs as the pre-failure stage. For each read, XFDetector first checks the *consistency state* and then the *persistence state* of the target location, as reading a consistent location is certainly bug-free, while reading a persisted location can still be semantically inconsistent. Reading a commit

[4]XFDetector also handles non-temporal writes and other types of fence.
[5]We reserve an extensibility as discussed in Section 3B.5.5 to support other crash consistency mechanisms.

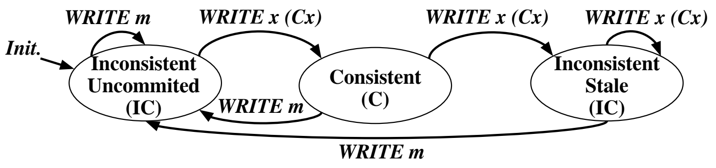
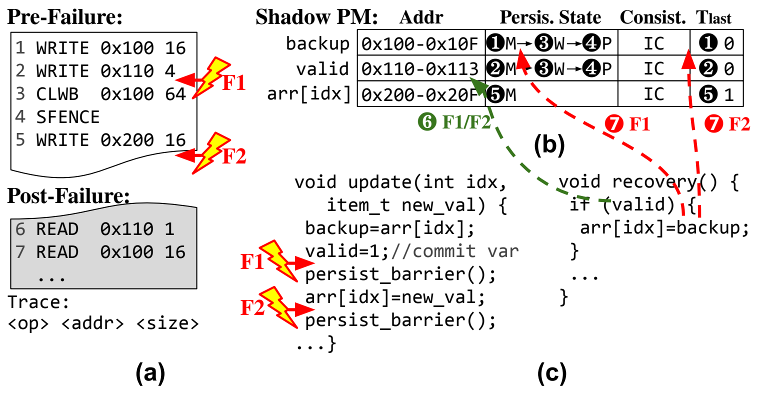


Figure 3B.10: Transitions of the consistency state.


Pre-Failure:

```
1 WRITE 0x100 16
2 WRITE 0x110 4
3 CLWB  0x100 64
4 SFENCE
5 WRITE 0x200 16
```

Post-Failure:

```
6 READ  0x110 1
7 READ  0x100 16
  ...
Trace:
<op> <addr> <size>
```

Shadow PM:	Addr	Persis. State	Consist.	Tlast
backup	0x100-0x10F	❶M→❸W→❹P	IC	❶ 0
valid	0x110-0x113	❷M→❸W→❹P	IC	❷ 0
arr[idx]	0x200-0x20F	❺M	IC	❺ 1

```
void update(int idx,
   item_t new_val) {
 backup=arr[idx];
 valid=1;//commit var
 persist_barrier();
 arr[idx]=new_val;
 persist_barrier();
...}
```

```
void recovery() {
 if (valid) {
  arr[idx]=backup;
 }
 ...
}
```

Figure 3B.11: (a) The pre- and post-failure traces, (b) the states in the shadow PM, and (c) the code demonstrating the steps of the detection procedure.

variable is a *benign cross-failure race* and not regarded as a bug. On detection of a cross-failure bug, XFDetector reports the file name and the line number of the reader and the last writer that cause the bug. Next, we illustrate the detection procedure with an example.

Example. Figure 3B.11 demonstrates the detection procedure, where Figure 3B.11a, 3B.11b, and 3B.11c show the pre- and post-failure trace, the shadow PM, and the code, respectively. The `valid` variable is marked as a commit variable because it decides the validity of the backup and the in-place update. XFDetector injects a failure point before each of the two ordering points (F1 and F2 before each `persist_barrier()`), and each failure point triggers their corresponding post-failure execution. We take the first two entries (line 6 and 7) from the post-failure trace for demonstration. Initially, the *global timestamp* is 0 and all PM addresses in the example are *unmodified*. Next, we demonstrate the detection step-by-step. Line 1: creates `backup` and updates its PM status to *modified*. Line 2: sets `valid` and updates its PM status to *modified*. As there is no update before the commit timestamp, XFDetector does not change the consistency state of any PM location. F1: the first failure triggers the post-failure execution. Line 6 (F1): reads from `valid` (the commit variable). Line 7 (F1): reads from `backup` for rolling back. However, as the PM state of `backup` is *modified*, XFDetector reports a cross-failure race. Then, it continues pre-failure execution from F1. Line 3: writes back a cache line that contains both `backup` and `valid`, and updates both PM status to *writeback-pending*. Line 4: places an `SFENCE` to make sure previous pending writebacks are complete, and increments the *global timestamp*. Line 5: updates the variable `arr` in-place and the persistence status becomes *modified*. F2: the second failure triggers the post-failure execution. Line 6 (F2): reads from `valid` (the commit variable). Line 7 (F2): reads from `backup` for rolling back. However, as the consistency state is *inconsistent*, XFDetector reports a cross-failure semantic bug. This bug is due to `backup` not being updated *before* the last update to the commit variable (`valid`). In summary, XFDetector reports a cross-failure race at the first failure point (F1), and a cross-failure semantic bug at the second failure point (F2).

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

Complexity. Assuming there are F failure points in the pre-failure stage, and each corresponding post-failure execution has P operations on average, the complexity of the detection procedure is $O(F \cdot P)$. We observe that the post-failure execution in most crash consistency mechanisms takes a small, constant number of steps to recover from the failure. For example, an undo logging mechanism only recovers the last incomplete transaction. Therefore, the detection time scales *linearly* with the number of failure points in most scenarios. We evaluate XFDetector's scalability in Section 3B.6.2.

3B.5.5 Extensibility

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

Extending Operation Tracing. XFDetector decouples the frontend tracing from the backend detection. The frontend of XFDetector is built on Intel's Pin [177] for fine-grained, automated tracing. Although Pin is limited to user-space programs and Intel processors, the backend of XFDetector

Table 3B.3: The evaluated system.

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

can be attached to other tracing frameworks, such as the software-directed tracing in WHISPER [1] and PMTest [25].

Extending Detection Capability. We summarize the possible approaches for extending XFDetector as the following points. First, XFDetector functions (Table 3B.2) can work as building blocks to support other PM libraries. Take our implementation as an example, we skip the detection of PMDK's internal transactions but instead explicitly add a failure point for each library function that contains ordering points. This way, XFDetector only needs to handle programmer's code. Second, if the target program applies a crash consistency mechanism that does not follow the approach described in Section 3B.3.2, programmers may need to modify the tool and provide extra annotations. For example, to support a version-based mechanism that does not take the latest copy but uses a specific one in the log, programmers need to add extra timestamps to track when the log was committed. The checksum-based mechanism is another example, where the consistency status is not determined by a commit variable but uses a pair of data and its associated checksum. To test the correctness of the checksum implementation, programmers may manually place a failure point using XFDetector's library function or modify the failure injection mechanism to automatically add more failure points between ordering points. Third, if a cross-failure bug is beyond the capability of XFDetector, the failure injection framework can work in cooperation with conventional debugging techniques. For example, bugs that depend on data values, such as creating a log using incorrect data, cannot be detected because XFDetector does not track data values. To detect such bugs, programmers may place assertions to check data values in the post-failure code and then use XFDetector's failure injection mechanism to trigger the post-failure execution.

Table 3B.4: The evaluated PM programs.

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

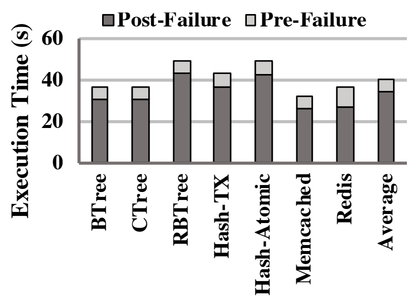
3B.6 Evaluation

3B.6.1 Methodology

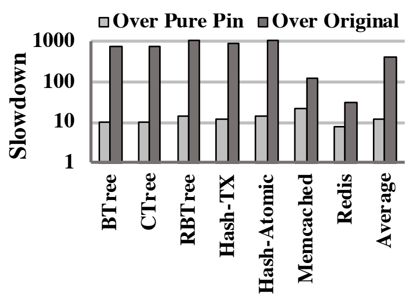
We evaluate our tool, XFDetector in a real system (Table 3B.3) with Intel's Optane DC Persistent Memory Module (DCPMM). PM is mounted with the DAX option to bypass OS indirections [178]. Table 3B.4 lists the evaluated PM programs, including 5 micro benchmarks from PMDK [35] examples and 2 real-world workloads: Redis [14] and Memcached [12]. The transaction-based programs are built with PMDK's libpmemobj, and the low-level ones are built with libpmem. We annotate the source code with XFDetector interface for cross-failure bug detection. Table 3B.4 lists the lines of code (LOC) of the original version and our annotation. We modify the `Makefile` to link the test program with the shared object of XFDetector's interface for all workloads. We mark the entire program as RoI (both pre- and post-failure) for the micro benchmarks, and select the code region that performs updates to PM objects as the pre-failure RoI and the region that performs recovery as the post-failure RoI for larger real-world workloads.

3B.6.2 Performance

Execution Time. This experiment evaluates the execution time of XFDetector. We run each workload with one transaction/query that performs an insertion, and another one for each failure point. Figure 3B.12a shows the wall-clock time (seconds) of XFDetector for each workload. XFDetector takes an average of 40.6 seconds to analyze one insertion operation. We further break down the execution time into two parts: the pre- and post-failure stages. We observe that the post-failure takes the majority of the execution time as XFDetector spawns the post-failure execution for each failure point. We further compare the execution time of XFDetector with a "Pure Pin" configuration where the Pintool only traces the PM read/write operations, and the original program that runs



(a)



(b)

Figure 3B.12: Performance of XFDetector: (a) wall-clock time and (b) slowdown over pure Pin and original program.

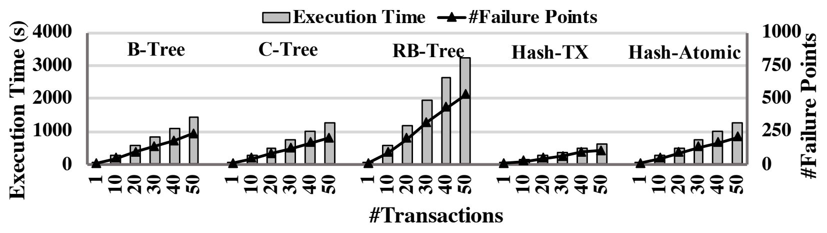


Figure 3B.13: The execution time of micro benchmarks with variable numbers of pre-failure transactions.

without any tool (Figure 3B.12b). On average (Geo. mean), XFDetector is 12.3× slower than "Pure Pin" and 400.8× slower than the original program. We conclude that the repeated post-failure execution is the major bottleneck, and Pintool is the secondary bottleneck. However, the post-failure executions are independent as they operate on a copy of the original PM image, and therefore, can be parallelized. We leave the parallelized detection as a future work.

Scalability. This experiment scales the number of transactions performed in the pre-failure stage during detection. As real-world workloads execute upon query, we scale the number of pre-failure transactions in micro benchmarks and keep the post-failure constant (one transaction). The primary axis in Figure 3B.13 indicates the execution time (wall-clock time) of detection with variable numbers of pre-failure transactions, and the secondary axis indicates the number of failure points in the pre-failure stage. This experiment shows that the execution time increases *linearly* as the number of failure points increases.

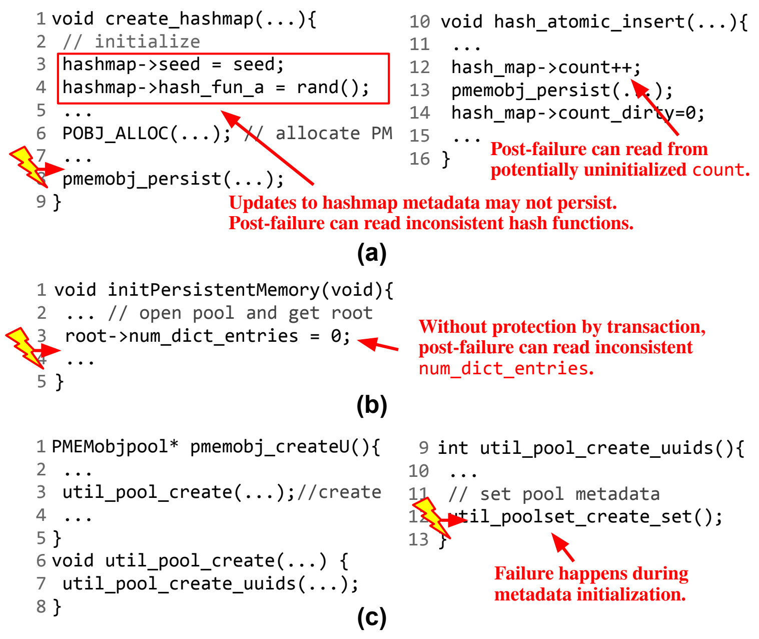
3B.6.3 Detection Capability

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

Validation. Table 3B.5 summarizes the synthetic bugs that we have validated using XFDetector. We first validate XFDetector's detection capability with the bug suite from PMTest [25]. As

Table 3B.5: The synthetic bugs for validation
(**R**: cross-failure race, **S**: cross-failure semantic bug, and **P**: performance bug).

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



```
1 void create_hashmap(...){
2  // initialize
3  hashmap->seed = seed;
4  hashmap->hash_fun_a = rand();
5  ...
6  POBJ_ALLOC(...); // allocate PM
7  ...
8  pmemobj_persist(...);
9 }
```

```
10 void hash_atomic_insert(...){
11  ...
12  hash_map->count++;
13  pmemobj_persist(...);
14  hash_map->count_dirty=0;
15  ...
16 }
```

Post-failure can read from potentially uninitialized count.
Updates to hashmap metadata may not persist.
Post-failure can read inconsistent hash functions.
(a)

```
1 void initPersistentMemory(void){
2  ... // open pool and get root
3  root->num_dict_entries = 0;
4  ...
5 }
```

Without protection by transaction, post-failure can read inconsistent num_dict_entries.
(b)

```
1 PMEMobjpool* pmemobj_createU(){
2  ...
3  util_pool_create(...);//create
4  ...
5 }
6 void util_pool_create(...) {
7  util_pool_create_uuids(...);
8 }
```

```
9 int util_pool_create_uuids(){
10  ...
11  // set pool metadata
12  util_poolset_create_set();
13 }
```

Figure 3B.14: New bugs detected by XFDetector in
(a) Hashmap-Atomic, (b) Redis, and (c) libpmemobj.

cross-failure semantic bugs are beyond PMTest's scope, we create additional synthetic, cross-failure semantic bugs on top of the Hashmap-Atomic example which is built on low-level primitives. We do not create cross-failure semantics bugs for other workloads as the commit variables are managed by their transactional library functions. We also create other cross-failure race bugs for better validation. The validation shows that XFDetector is effective in detecting these synthetic bugs and covers more types of bugs than existing works [25, 128].

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

3B.7 Discussion

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

Detection Scope. XFDetector can detect cross-failure bugs due to reading non-persisted or semantically inconsistent data. The detection mechanism takes into account the address and the order of PM updates instead of data values (except for commit variables that can affect the procedures in the post-failure stage). Therefore, programming errors such as writing incorrect data values to non-commit variables (e.g., log incorrect data) are out of the scope. Section 3B.5.5 has described the way to extend the capability by incorporating conventional debugging methods with XFDetector.

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

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

Chapter 3C

Test Case Generation for Persistent Memory Programs

3C.1 Introduction

So far, we have introduced two testing frameworks. In Chapter 3A, we introduced our runtime testing framework, PMTest [25], that detects violations against the ordering and persistence requirements. In Chapter 3B, we introduced an end-to-end testing framework, XFDetector [26], that detects the correct failure-recovery behavior by introducing failures during the execution. However, there is another major issue remains unsolved—to detect a crash consistency bug, the *buggy procedure needs to be executed*. For example, to reproduce a bug in PMDK [134] that was reported by PMTest [25], the inputs to a B-Tree-based key-value store need to be carefully designed, in order to execute a program path that triggers B-Tree's insertion and rebalancing procedures. Hence, even with the aid of PM testing tools, bugs cannot be detected without having inputs to trigger the required execution path. In this chapter, we aim to assist PM programming by *generating test cases* to cover nontrivial crash consistency and performance bugs.

Due to the already complicated programming for PM systems, a tool for test case generation ideally should not place an additional burden on programmers. *Fuzzing*, a widely-used test case generation method, perfectly satisfies this demand as it requires minimum knowledge about the target code base and has been proven to be effective [179–183]. At a high-level, a fuzzer *iteratively* generates new test cases by mutating existing ones, where high-value test cases, such as those that explore new branches, are reused in future iterations. Although fuzzing is an effective method, we identify that in order to generate test cases for PM programs efficiently, additional requirements need to be satisfied.

First, PM programs maintain the persistent state on PM devices (e.g., as a PM image in a DAX file system), different from conventional programs. A PM program takes not only the *regular program input* (e.g., a command that inserts a key-value pair) but also a *PM image* which contains an existing persistence state. As the procedure of loading an existing PM image and performing operations on top can also face crash consistency bugs [26, 184], it is necessary for a fuzzer to provide PM images as inputs. Fuzzers for conventional programs perform mutation to generate regular inputs (e.g., commands). In comparison, PM images have a much larger exploration space (e.g., tens of MBs). Therefore, generating PM images through direct mutation is ineffective and will likely produce invalid images. For example, a randomly mutated PM image may have illegal pointers that may cause the program to abort in the beginning without exploring any useful paths. Even though recent works have designed fuzzers for file system images, they require a well-defined image layout [185, 186]. As PM programs tend to customize the persistent data management, methods taken by file system fuzzers are not suitable for PM image generation. Therefore, the *first challenge* is to efficiently generate *valid PM images*.

Second, PM programs also need to recover from PM images that are resulted from failures during program execution, which we refer to as *crash images*. Prior works have shown that the recovery procedure is also susceptible to crash consistency bugs [26, 184]. Therefore, the fuzzer needs to generate not only *normal PM images* but also *crash images* for thorough testing. However, a program can fail at any point during execution, leading to a potentially infinite number of crash images. Therefore, the *second challenge* is to generate *crash images* that are most effective for testing.

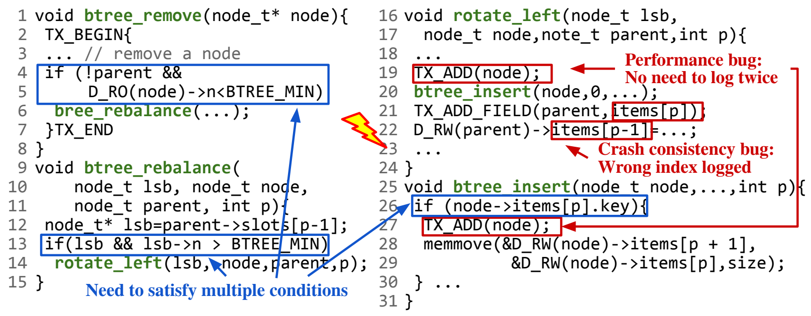
Finally, PM programs may contain procedures for different purposes, not limited to managing PM, especially in real-world workloads. On the other hand, only PM operations are critical to crash consistency bugs—performing writes to PM without taking care of their ordering can leave inconsistent data on PM, and reading from them can cause the later execution to behave incorrectly [26]. However, traditional coverage metrics, such as branch coverage, used by conventional fuzzers do not target procedures with the most concerned PM operations. Therefore, the *third challenge* is to design a fuzzer that can *target PM-related procedures*.

The new requirements for test case generation are critical to systematically testing PM programs. However, existing fuzzers are incapable of meeting these requirements. In this paper, we develop PMFuzz (available at `https://pmfuzz.persistentmemory.org`), a fuzzer that aims to generate test cases for detecting crash consistency and performance bugs in PM programs. Next, we describe the three high-level ideas of our design.

PM Image Generation. Existing fuzzers either do not target large PM images or require a fixed image layout, as directly mutating an image can likely generate invalid images that cannot explore useful paths. Therefore, an effective image generation method should guarantee valid PM images. We observe that a PM image is essentially an outcome of input commands. Therefore, our key idea is to leverage the program logic to *mutate* an existing PM image. PMFuzz incrementally generates the image by applying the fuzzing logic on the input commands. And eventually, the PM image will be thoroughly mutated through the iterative fuzzing procedure.

Crash Image Generation. In addition to taking *normal images* as inputs, PM programs can also execute on *crash images* that are caused by failures. Although a failure can occur at any point during execution, the recovery procedure typically depends on a few key variables that are stored in the image. For example, an undo-log-based program performs the following steps: back up the old data in the undo log, set the valid bit of the log, perform in-place update, and finally unset the valid bit. In case of a failure, the recovery procedure will take one of these two paths depending on the value of the valid bit: one path applies the undo log and the other directly resumes the execution. As such, there is a *control-flow dependency* between the execution before and after the failure. Based on this dependency, only two failure images are needed to cover both paths: one with the valid bit set to one and another set to zero. Our key idea is to minimize the number of crash images by only generating the images that can affect the control-flow in the recovery procedure.

Coverage of PM Path. As crash consistency and performance bugs are caused by the misuse of PM operations, achieving high coverage of these bugs requires the fuzzer to perform a *targeted fuzzing* on program paths with PM operations. To enable this prioritization, we first define the *PM path* as a path that consists of program statements with PM operations (e.g., read, write, writeback, etc.). Then, PMFuzz monitors the statistics of PM paths during fuzzing, and prioritizes test cases



```
void btree_remove(node_t* node){
 TX_BEGIN{
 ... // remove a node
 if (!parent &&
      D_RO(node)->n<BTREE_MIN)
  bree_rebalance(...);
 }TX_END
}
void btree_rebalance(
    node_t lsb, node_t node,
    node_t parent, int p){
 node_t* lsb=parent->slots[p-1];
 if(lsb && lsb->n > BTREE_MIN)
  rotate_left(lsb, node,parent,p);
}
```

```
void rotate_left(node_t lsb,
  node_t node,note_t parent,int p){
 ...
 TX_ADD(node);
 btree_insert(node,0,...);
 TX_ADD_FIELD(parent,items[p]);
 D_RW(parent)->items[p-1]=...;
 ...
}
void btree_insert(node_t node,...,int p){
 if (node->items[p].key){
  TX_ADD(node);
  memmove(&D_RW(node)->items[p + 1],
           &D_RW(node)->items[p],size);
 } ...
}
```

Figure 3C.1: A buggy PM-based B-Tree (Example 1).

that cover *new* PM paths. By focusing on PM paths, PMFuzz can efficiently generate more test cases that target crash consistency and performance bugs.

Based on the key insights above, we implement PMFuzz on top of an open-source fuzzer, AFL++ [182], and evaluate it in a real PM system. Our contributions are the following:

- PMFuzz is the first test case generator for detecting crash consistency and performance bugs in PM programs.

- We evaluate PMFuzz using eight representative PM programs in a real PM system. On average, PMFuzz covers 4.6× more PM paths over the well-known fuzzer, AFL++, within 4 hours of fuzzing.

- Even though these PM programs have been extensively tested by prior works [25, 26, 128, 129], we detect 12 new real-world bugs with PMFuzz's systematic test case generation.

3C.2 Background and Motivation

Chapter 3A and 3B have described two testing frameworks that detect crash consistency bugs. Both tools require that the program path that leads to the crash consistency bug needs to be executed in order to detect such a bug. However, it is not always trivial to trigger a buggy program path. Next, we show two examples of non-trivial bugs.

3C.2.1 Nontrivial Bugs in PM Programming

Example 1: A buggy B-Tree. Figure 3C.1 (Example 1) shows a simplified code snippet of a B-Tree that is implemented with PMDK's transaction library. The `btree_remove()` and `btree_insert()` procedures are wrapped inside a pair of `TX_BEGIN` and `TX_END` to ensure a consistent recovery after failure. Within the procedure, `TX_ADD()` is used to make a backup of the persistent data before it is modified. B-Tree is a commonly-used structure for key-value stores, where each node contains a number of keys. To remove an existing key from a B-Tree, the program first calls `btree_remove()`. After removal, if the number of keys (`n`) becomes less than `BTREE_MIN`, it rebalances the tree by calling `btree_rebalance()` (line 4-6), which left-rotates the modified node if the number of keys in its left sibling (`lsb`) exceeds `BTREE_MIN` (line 13-14). During the rotation process, `rotate_left()` calls the insertion function `btree_insert()` (line 18), which then checks the validity of the key (line 23), and performs the rotation (line 28-29). Finally, after insertion, `rotate_left()` updates `items` in its `parent` node (line 21-22).

Although this example seems to be correct as the whole procedure is wrapped in a transaction, there are two bugs. The first one is a crash consistency bug, where the program updates the `(p-1)`-th `item` (line 22) but logs the `p`-th `item` by mistake (line 21). In case of a failure at line 22, the `item` being modified can be lost as it has not been backed up by the log. The second one is a performance bug, where `rotate_left()` and `btree_insert()` attempt to log the same node twice (line 19 and 27), leading to unnecessary performance degradation.

These bugs in Example 1 have one major similarity that is they cannot be directly observed by programmers. A crash consistency bug, such as incorrect ordering or backup, does to affect the current volatile state, thus is not visible until a failure occurs during the buggy procedure. And, a performance bug, such as using excessive ordering or unnecessary logging, does not affect the ongoing execution. To make these bugs visible to programmers, there have been tools tailored for PM programming [25, 26, 128, 129]. These tools keep track of PM operations at runtime, and then detect violations against the crash consistency guarantees. These tools have the capability of detecting the bugs in Example 1. Nonetheless, they all *require the buggy program path to be executed* in order to detect the violations. In Example 1, the program needs to satisfy two `if` conditions to detect the crash consistency bug (line 21-22). Even harder, triggering the performance bug (line 27) requires satisfying all three `if` conditions. Therefore, a test case generator becomes a necessity to cover such nontrivial program paths. Next, we introduce *fuzzing*, a widely-used technique for test case generation.

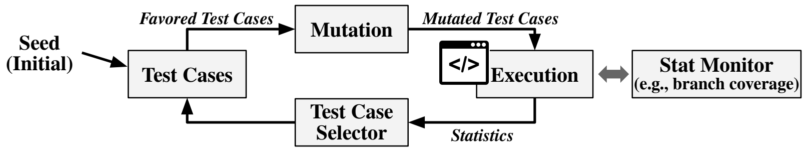
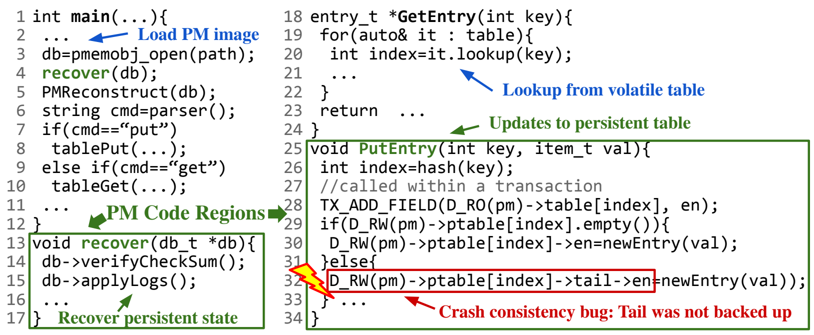


Figure 3C.2: A general fuzzing procedure.



```
 1 int main(...){
 2  ...
 3  db=pmemobj_open(path);
 4  recover(db);
 5  PMReconstruct(db);
 6  string cmd=parser();
 7  if(cmd==“put”)
 8   tablePut(...);
 9  else if(cmd==“get”)
10   tableGet(...);
11  ...
12 }
13 void recover(db_t *db){
14  db->verifyCheckSum();
15  db->applyLogs();
16  ...
17 }
18 entry_t *GetEntry(int key){
19  for(auto& it : table){
20   int index=it.lookup(key);
21   ...
22  }
23  return  ...
24 }
25 void PutEntry(int key, item_t val){
26  int index=hash(key);
27  //called within a transaction
28  TX_ADD_FIELD(D_RO(pm)->table[index], en);
29  if(D_RW(pm)->ptable[index].empty()){
30   D_RW(pm)->ptable[index]->en=newEntry(val);
31  }else{
32   D_RW(pm)->ptable[index]->tail->en=newEntry(val));
33  } ...
34 }
```

Load PM image
PM Code Regions
Recover persistent state
Lookup from volatile table
Updates to persistent table
Crash consistency bug: Tail was not backed up

Figure 3C.3: A buggy PM-based database (Example 2).

3C.2.2 Requirements for Fuzzing PM Programs

A test case generator for testing PM programs should avoid introducing additional burdens on programmers, given the already complicated nature of PM programming. *Fuzzing* is a well-known technique that automatically generates test cases while minimizing programmers' effort [179–183]. Figure 3C.2 shows a typical procedure of fuzzing—a fuzzer takes a set of initial test cases (or seeds), performs mutation on those test cases, executes the target program, monitors the execution statistics, and finally uses the statistics (e.g., branch coverage) to select high-value test cases. These high-value test cases will then be used in the next iteration of fuzzing. Using a fuzzer, the if-conditions in Figure 3C.1 (Example 1) are likely to be covered. However, we identify that there are additional needs from PM programs that conventional fuzzers do not meet. Next, we provide another example of a PM crash consistency bug to motivate the new requirements.

Example 2: A buggy PM database. Figure 3C.3 (Example 2) is a simplified example of a database based on the PMDK transaction [35]. It maintains the persistent data in PM and buffers a volatile table in DRAM for faster lookup, similar to the PM-based Redis [14]. During execution, the `main()` function first loads the existing persistent data that were stored on PM, which we refer to as a *PM image* (line 3), calls `recover()` to restore the persistence state (e.g., recover from a previous failure), and then loads the PM structures to the volatile table. Upon requests, the database calls corresponding functions, such as `GetEntry()` and `PutEntry()`. `GetEntry()` (line 18) looks up the `key` in the volatile table, and `PutEntry()` (line 25) updates the `key-value` pair in the persistent

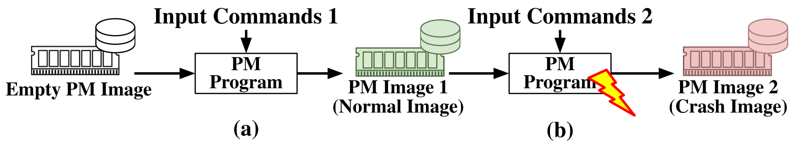


Figure 3C.4: PM program execution procedures that generate (a) a normal image, and (b) a crash image.

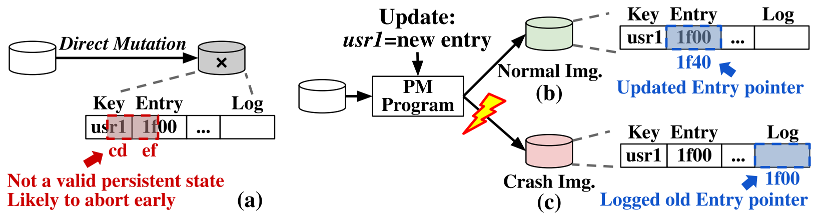


Figure 3C.5: (a) An invalid image produced by direct mutation, (b) a normal image produced by program logic, and (c) a crash image produced by program logic.

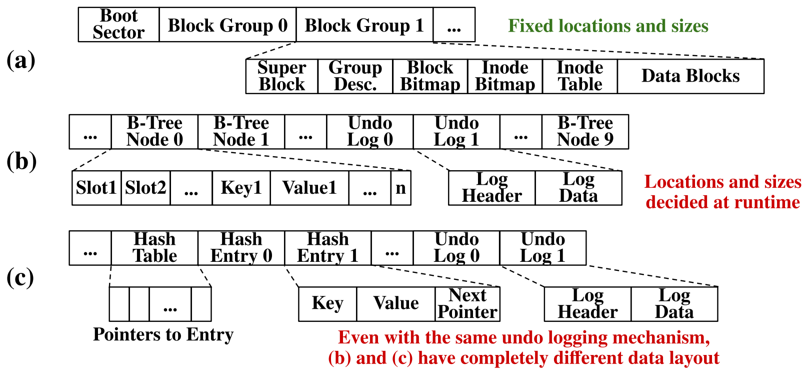


Figure 3C.6: Persistent data layout in (a) an Ext2 file system [4], (b) a PM-based B-Tree, and (c) a PM-based database.

`ptable`. In this example, there is a crash consistency bug in `PutEntry()`. A new entry is appended to the `tail` of the indexed list in `ptable` when the list is not empty (line 32), whereas the previous log operation only covers the first item in the list (line 28). Thus, in case a failure happens at line 32, the update to `tail` can be interrupted and remains in an inconsistent state. Next, we summarize the additional requirements that traditional fuzzers need to expose PM bugs.

Requirement 1: PM images as input. A PM program typically takes PM image(s) as part of the input to maintain their persistent state, as demonstrated by the procedure in Figure 3C.4a, and the `main()` function of Figure 3C.3 (Example 2). Prior works have shown that the procedure that loads PM images can be buggy [26]. Therefore, a fuzzer for PM programs needs to generate not only the basic input commands but also PM images for testing. More importantly, the generated PM image is required to be *valid*, so that the program can execute a useful path, without failing basic image checks or triggering exceptions. However, directly fuzzing PM images through mutation is challenging—the search space of a PM image (tens of MBs) is huge, and it is hard to construct a valid PM image. Figure 3C.5a demonstrates a PM image of a database being randomly mutated, where the mutation lies in the middle of the key and its entry pointer. Execution using this invalid image is likely to abort due to segmentation faults. Recent fuzzers have proposed to mutate file system images [185, 186] based on the preknowledge of the data layout of file systems. Figure 3C.6a shows the simplified layout of an Ext2 file system [4], where the sizes and locations are known based on the Ext2 format. In comparison, PM programs tend to customize the way they manage persistent data. Figure 3C.6b demonstrates the layout of Example 1, where the structures of tree nodes and logs are seemly rigid but do not follow a specific format—the nodes and undo-log entries are all allocated in the image at runtime. Figure 3C.6c shows the layout of Example 2. Despite the use of a similar undo-logging mechanism, the data layout still differs from that of Example 1, due to their fundamental algorithmic differences.

Requirement 2: Crash images as input. PM programs are expected to be recoverable from unexpected failures. Thus, they may also load PM images caused by failures. For clarity, we refer to a PM image that is an outcome of an *uninterrupted* execution as a *normal image*, and an image that results after a *failure* as a *crash image*. Figure 3C.4b shows a procedure, where a PM program takes an existing PM image and executes a series of input commands. During execution, a failure occurs and results in a crash image. After the program restarts after the failure, it needs to execute the recovery procedure. For example, Figure 3C.3 (Example 2) validates the image checksum (line 14) and rolls back the prior updates using the logged data (line 15). In order to detect bugs during the recovery procedure, a crash image is also a necessity for the input test case. However, failures may happen at any point during execution, and therefore, can lead to an infinite number of crash images.

Requirement 3: Targeting PM operations. The crash consistency bugs and performance bugs are caused by PM operations, such as PM writes that modifies the state, and PM reads that loads an existing state [26]. Therefore, test case generation should be focused on program paths that perform PM operations. In real-world PM programs, such as database applications, there are both volatile and persistent code regions. In Figure 3C.3 (Example 2), only a fraction of the code is performing PM operations, as marked by the green boxes. As such, a fuzzer should ideally focus on the interesting paths with PM operations. However, traditional coverage metrics, such as branch coverage, which are widely adopted by traditional fuzzers do not target these PM-related paths.

3C.3 High-Level Design of PMFuzz

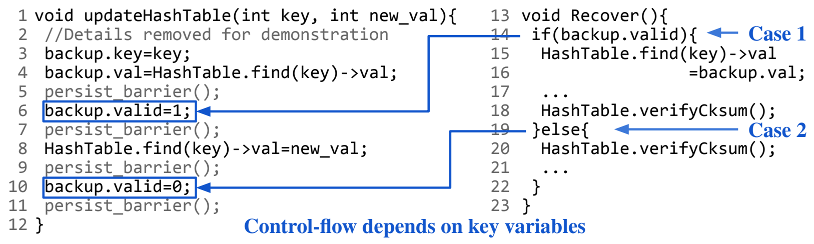
So far, we have described the new requirements for fuzzing PM programs. In this work, we propose PMFuzz, a fuzzer that aims to efficiently generate test cases for debugging PM programs. Next, we discuss the challenges and our high-level design.

3C.3.1 Normal PM Image Generation

Challenge. PM programs require that a fuzzer generates valid PM images to explore useful program paths. Conventional fuzzers are only capable of fuzzing small inputs thus do not meet this requirement. Even though file system fuzzers target large file system images, they require a well-formulated rule and image layout [185, 186]. In comparison, a PM image is not only large (e.g., tens of MBs) but also highly customized. Thus, fuzzing PM images is beyond the capability of existing fuzzers. Therefore, the *first challenge* is how can PMFuzz *efficiently generate PM images*?

Observation. As the data layout of a PM program can be largely customized, directly generating a valid PM image with permutation is hard. However, the outcome of the program logic itself always results in a *valid* persistent state. As Figure 3C.4 demonstrates, the PM program incrementally mutates the PM image with input commands. Therefore, instead of directly fuzzing the PM image, a more effective alternative is to indirectly fuzz the input commands, which in turn will mutate the image from one valid state to another.

Solution. Based on this observation, our key idea is to fuzz the input commands and reuse the program logic to generate a PM image that is guaranteed to be a valid persistent state. At the high-level, the procedure of fuzzing PM images follows these steps: (1) Mutate input commands, (2) perform execution on top of an existing PM image, (3) collect the output PM image, and (4) reuse the generated PM images and repeat these steps. As PMFuzz continues to recursively operate on existing PM images, a thorough mutation on the PM image will eventually be done by the program logic itself. Figure 3C.5b demonstrates that executing an update command creates an output PM image that has a valid mutation on the value of "Entry pointer". Thus we conclude that leveraging program logic can efficiently generate valid PM images.



```
1 void updateHashTable(int key, int new_val){
2  //Details removed for demonstration
3  backup.key=key;
4  backup.val=HashTable.find(key)->val;
5  persist_barrier();
6  backup.valid=1;
7  persist_barrier();
8  HashTable.find(key)->val=new_val;
9  persist_barrier();
10 backup.valid=0;
11 persist_barrier();
12 }
```

```
13 void Recover(){
14  if(backup.valid){
15   HashTable.find(key)->val
16                  =backup.val;
17   ...
18   HashTable.verifyCksum();
19  }else{
20   HashTable.verifyCksum();
21   ...
22  }
23 }
```

Figure 3C.7: Example of control-flow dependency between failures and the recovery procedure.

3C.3.2 Crash Image Generation

Challenge. As PM programs are expected to recover from failures, they may also take *crash images* as the input. However, there can be an infinite number of crash images because failure can happen at any point in the program. Thus, the *second challenge* is how PMFuzz can *generate crash images* that are most effective?

Observation. Figure 3C.7 shows an example of updating a hash table using low-level PM primitives. The program first backs up the existing `key` and `value` (line 3-4), sets the `backup` to be valid (line 6), performs the in-place update in the destination entry (line 8), and finally invalidates the `backup` (line 10). In case this procedure is interrupted by a failure, the program has a `recovery()` function. If the `backup` is valid (line 14), it rolls back the updates (line 15-16) and then verifies the checksum of the hash table (line 18). Otherwise, it verifies the checksum directly (line 20). Given a crash image that is generated during the procedure of `updateHashTable()`, the two paths during `recovery()` (as indicated by Case 1 and 2) only depend on the value of `backup.valid`. Therefore, even though a failure can happen at any point during the execution, not all resulting crash images are important for the coverage.

Solution. Inspired by the prior works that model the relationship between PM program recovery and failures [26, 159, 163, 164], we model the relationship between the program path during recovery and the prior procedure during the normal execution as a *control-flow dependency*. The significance of a crash image boils down to whether it can lead to a persistent state that affects the *control-flow* in the procedure after failure. Updates that can lead to a different control-flow are typically applied to key variables that determine the consistency state. For example, the update to `backup.valid` in Figure 3C.7 alters the consistency state. Other examples include commit bits in undo/redo logs, and timestamps in checkpointing mechanisms. Usually, updates to such a commit variable are wrapped with *ordering points* (e.g., using a `persist_barrier()`), such that the commit variable always persists after the prior PM updates but before the successive ones.

Following this observation, our approach that reduces the number of crash images is two-fold. First, PMFuzz focuses on placing failures at ordering points to reduce the number of failure images. Second, PMFuzz also places additional failure points probabilistically, at a configurable rate. This way, even if the program is completely buggy, i.e., with a large number of misplaced ordering points, PMFuzz will still generate failure images for debugging. In both cases, crash images are generated by interrupting the execution of input commands. Therefore, all crash images maintain valid persistent states of the program. Back to the example in Figure 3C.5, by placing a failure at the point where an undo log of the entry has been persisted but the item has not been updated, the output image will contain the old value in the "Log entry" of the crash image. During the recovery procedure, the program will use this "Log entry" to reconstruct the table.

3C.3.3 Coverage for PM Path

Challenge. PM programs can contain various procedures but only those with PM operations can lead to crash consistency and performance bugs. The *third challenge* is how can PMFuzz efficiently generate test cases that *target PM operations*?

Observation. As prior testing works for PM programs [25, 26, 128, 184] have shown, crash consistency bugs (and also performance bugs) occur due to inappropriate PM accesses. Therefore, PMFuzz should target code regions that perform PM operations, E.g., PM reads, writes, writeback/flush primitives, and fences. However, PM reads and writes cannot be easily distinguished from regular volatile ones as they only differ in the address. Prior testing tools have been using dynamic instrumentation to keep track of these operations at the cost of tens- to hundreds-time overhead [26, 128, 129, 184]. As one of the key design principles of fuzzing is to achieve high execution efficiency, dynamic instrumentation is not a feasible choice. Despite the difficulties, we find that it is not necessary to track at the instruction granularity; instead, accesses to PM are typically wrapped with functions. PM libraries tend to restrict the way programs interact with PM. For example, the transaction library from Intel's PMDK [35], libpmemobj [187], exposes `D_RO` and `D_RW` (direct read-only/read-write) functions to obtain pointers to objects in PM. And, the lower-level PM library, libpmem [176], also provide methods, such as **`pmem_persist()`**, to write-back persistent data. Therefore, the tracking granularity can be lifted to the function-level to reduce the performance overhead.

Solution. Based on the two observations, our key idea is to identify PM operations by tracking them at the granularity of PM library functions. Having PM operations being tracked, we can

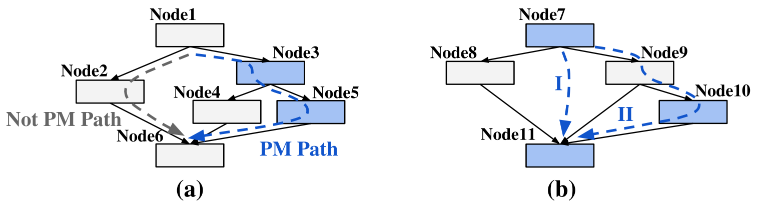


Figure 3C.8: PM path examples (nodes in *blue* are PM nodes).

further design a PM-specific coverage metric to enable a *targeted fuzzing* on the PM-related program paths (see Section 3C.4.2 for details about the mechanism). Next, we formally define the program path that contains PM operations.

- ***Control-flow Graph (CFG).*** A CFG of a program procedure is a directed graph represented by a tuple of $\langle N, E\rangle$; N is the set of *nodes*, where each node n represents unique program statement; $E \subset N \times N$ is the set of *edges*, where an edge e_{ij} represents execution flow between nodes n_i and n_j.

- ***Program Path (π).*** A program path in a CFG is a sequence of nodes $\pi = \langle n_0, n_1, ...\rangle$, such that there is an edge along the CFG between two consecutive nodes of the sequence.

- ***PM Node (p).*** A CFG node $p \in N$ is a PM node if it performs *at least one* PM operation.

- ***PM Path (π_{PM}).*** A PM path is a PM node sequence $\pi_{PM} = \langle p_0, p_1, ...\rangle$, such that, there is at least one edge along the CFG between two consecutive PM nodes in the sequence.

Figure 3C.8 shows two example CFGs, where nodes in blue are PM nodes that have PM operations. Based on the definitions above, in the CFG of Figure 3C.8a, the path of Node 1-2-6 is not a PM path due to the absence of PM operations, but the path of Node 1-3-5-6 does as it contains an edge between PM Node 3 and 5. In Figure 3C.8b, the path of Node 7-8-11 and Node 7-9-11 are regarded as the same PM path (marked as PM Path I), because they share the same PM nodes. In comparison, the path of Node 7-9-10-11 is unique because it contains a new PM Node, Node 10 (marked as PM Path II). By tracking PM paths, PMFuzz prioritizes test cases that explore new PM paths. Therefore, PMFuzz can more efficiently generate test cases for detecting crash consistency and performance bugs.

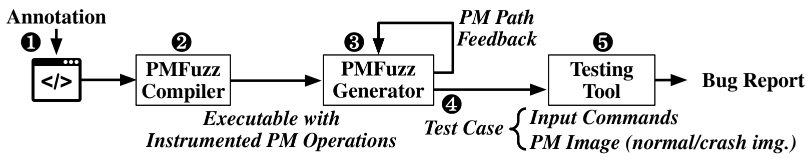


Figure 3C.9: High-level workflow of PMFuzz.

3C.4 Implementation of PMFuzz

In this section, we first present an overview of PMFuzz's workflow and then describe the details about the implementation.

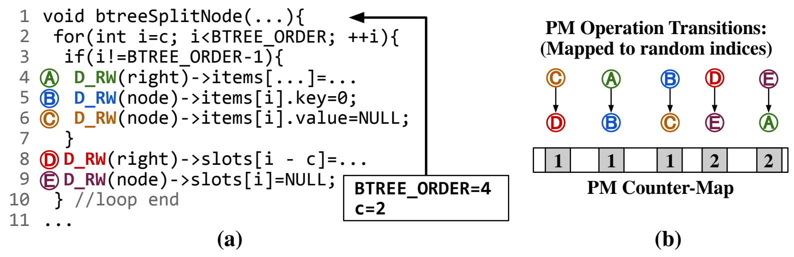
3C.4.1 Overview

PMFuzz is developed on top of a well-known fuzzer AFL++ [182]. It generates test cases to cover crash consistency and performance bugs in PM programs. Figure 3C.9 shows the high-level workflow. First, PMFuzz compiler instruments the source code to keep track of PM operations (step ❶ and ❷). Then, PMFuzz takes the compiled program and performs fuzzing. The fuzzing procedure executes multiple instances of the PM program for better efficiency. During the execution of each program instance, PMFuzz monitors the coverage of the PM path and provides feedback to the fuzzing logic such that it can target PM-related operations (step ❸) that are most critical to crash consistency bugs. After completing the execution of an instance, it saves the generated test case if it has explored a *new* PM path (step ❹). Each test case contains input commands and a PM image (both normal and crash images). Finally, PMFuzz sends the test cases to a testing tool (e.g., XFDetector [26] or Pmemcheck [128]) for bug detection (step ❺).

3C.4.2 PM Operation Tracking

PMFuzz focuses on generating test cases that cover program paths that contain PM operations, such as read/write accesses, and writeback and fence primitives. As Section 3C.3.3 has introduced, PMFuzz tracks these operations at the granularity of PM library functions. To enable this tracking, PMFuzz first performs static instrumentation using PMFuzz's compiler pass (based on LLVM [144]) and then tracks them dynamically during runtime. Next, we describe these two steps in detail.

(1) Static Instrumentation. PMFuzz tracks PM operations at function-granularity. We take an approach similar to Intel's Valgrind tool, Pmemcheck [128] and place PM operation hints inside the PMDK library. As programmers are typically agnostic about the low-level library implemen-



```
 1 void btreeSplitNode(...){
 2  for(int i=c; i<BTREE_ORDER; ++i){
 3   if(i!=BTREE_ORDER-1){
 4 Ⓐ  D_RW(right)->items[...]=...
 5 Ⓑ  D_RW(node)->items[i].key=0;
 6 Ⓒ  D_RW(node)->items[i].value=NULL;
 7   }
 8 Ⓓ D_RW(right)->slots[i - c]=...
 9 Ⓔ D_RW(node)->slots[i]=NULL;
10  } //loop end
11 ...
```

Figure 3C.10: (a) Code instrumentation, and (b) the corresponding state of the PM counter-map for tracking PM operations.

tation, this approach does not require any modification to programmers' application code. More specifically, PMFuzz tracks `libpmem` [176] functions that perform low-level PM operations, as well as `libpmemobj` [187] functions that provide the transaction interface. We also develop a compiler pass to support custom PM libraries. Users only need to annotate the declaration of each PM-operation function, and the compiler pass will automatically instrument the application code. Then, PMFuzz compiles the PM program and inserts a tracking function before each PM operation (i.e., library function's call site). Each tracking function is associated with a *unique ID* that marks its PM operation. Figure 3C.10a demonstrates a simplified `btreeSplitNode()` function that highlights five PM operations, and marks their IDs with circled-letters. Next, we describe how PMFuzz keeps track of the path at runtime using the unique ID of PM operations.

(2) Dynamic Tracking. A PM path consists of a series of transitions between PM operations. Inspired by the way AFL [179] tracks branches, PMFuzz encodes the transition between two PM operations based on their unique IDs, and updates a *PM counter-map* according to the encoded value of this transition. Algorithm 1 demonstrates the transition encoding and PM counter-map update. First, the tracking mechanism reads the current PM operation's ID (`curID`), which has been assigned during compile-time (line 3). Second, it encodes the transition from the previous PM operation (with `prevID`) to the current one by XORing the two IDs (line 4). This way, a transition is encoded as an ID that serves as the index (`loc`) to a PM counter-map. The counter indicates the number of visits of this transition, as every visit of this transition increments this counter value by 1 (line 5). For lower storage overhead, each counter value is encoded with an 8-bit integer. Third, to preserve the direction of this transition, the tracking mechanism right-shifts the `curID` by 1 bit before moving toward the next PM operation (line 6). Figure 3C.10b shows the state of a PM counter-map after `btreeSplitNode()` completes the for-loop (line 2-10), using input arguments listed in the text box. Next, we describe how PMFuzz's fuzzing logic monitors the statistics of the PM path.

Algorithm 1: Update to PM counter-map

```
begin updatePMCounterMap($Op, PMCounterMap$)
    if $Op \in PMOps$ then                          // When Op is a PM operation
        $curID = Op.ID$                             // Get ID of the current OP
        $loc = curID \oplus prevID$                 // Encode transitions between OPs
        $PMCounterMap[loc] + +$                     // Increment counter
        $prevID = curID \gg 1$                      // Right-shift one bit to track direction
    return $PMCounterMap$
```

Algorithm 2: PM path prioritization

```
begin PMPathFeedback($TestCase$)
    foreach $loc \in PMCounterMap$ do
        if $unseen(PMCounterMap[loc])$ then
            $Favored = 2$                           // High priority
        else if $diffCounter(PMCounterMap[loc])$ then
            $Favored = 1$                           // Medium priority
        else
            $Favored = 0$                           // Low priority
        $TestCase.Favored = Max(Favored, TestCase.Favored)$
    return $TestCase$
```

3C.4.3 Fuzzing Feedback Logic

The core fuzzing algorithm of PMFuzz provides feedback for future test case generation in order to optimize PM path coverage based on the statistics. As PMFuzz is built on top of AFL++ [182], we take a similar approach as AFL++, where we prioritize branch coverage, but also integrate an additional targeted fuzzing algorithm for PM operations. Algorithm 2 presents the prioritization algorithm of PMFuzz, which examines each location in the *PM counter-map* and sets the `Favored` value of the corresponding test case. Test cases with *unseen* PM counter-map locations are set as *high-priority*, those with significantly different counter values are set as *medium-priority*, and the remaining ones that are identical or with minor counter value differences are treated as *low-priority*. After each iteration of fuzzing, PMFuzz discards low-priority cases unless AFL++'s branch coverage logic favors them. In the next iteration of fuzzing, test cases with higher priority are more likely to be mutated to generate new test cases. This algorithm is effective but requires zero-randomness during execution, i.e., the same test case always produces the same path and PM image. Otherwise, the feedback on PM path coverage is unstable and the fuzzing outcomes are not reproducible. Next, we describe the derandomization approach.

3C.4.4 Execution Derandomization

As stated above, we notice that PM programs generally have nondeterministic execution due to three major sources of randomness. PMFuzz mitigates the randomness in the following approaches.

(1) UUID of PM Images. Each PM image created by the PMDK library [35] is associated with a *universally unique identifier* (UUID). The UUID is randomly generated during the image creation time. Therefore, it is hard to determine whether two PM images are generated from the same input or not, as the UUID in each PM image is always unique. We eliminate this randomness by overloading the UUID assignment function in PMDK (also extensible to other libraries) with our version that sets the UUID to a constant value.

(2) Address Randomization. The address randomization mechanism for both volatile and persistent addresses is another source of randomness. First, volatile addresses are randomized by the *address space layout randomization* (ASLR) technique. Because PM images may keep these random volatile pointers for convenience, we disable ASLR in the Linux kernel [188]. This method makes sure that the volatile pointers would not introduce randomness to PM images. Second, persistent addresses are randomized when the PMDK library maps a PM image to the virtual address space. We derandomize the persistent addresses by setting PMDK's environment variable `PMEM_MMAP_HINT` that forces the PM image to be mapped to the *same* virtual address every time it executes [176].

(3) External Randomness. Not only PM programs but their dependent external libraries also use time-dependent or other nondeterministic random number generators. Due to time-dependent randomness, the same input test case can lead to different execution paths. We remove this source of randomness by loading the Preeny library [189] before fuzzing. Preeny overwrites the calls to random number generators using its `derand` module, making sure that the random numbers remain the same in each run.

3C.4.5 Detailed Fuzzing Procedure

Figure 3C.11 demonstrates the fuzzing procedure. First, PMFuzz spawns several instances of the annotated PM program with seed test cases (step ❶). For each instance, it tracks the PM path at runtime. Upon observing a new PM path, it saves this test case for further PM image generation, and provides positive feedback to the input command fuzzing logic as described in Section 3C.4.3 (step ❷). In the PM image generation procedure, PMFuzz generates two types of PM images: normal images

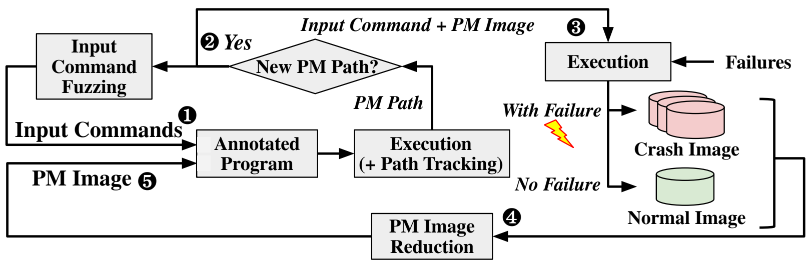


Figure 3C.11: Fuzzing procedure of PMFuzz.

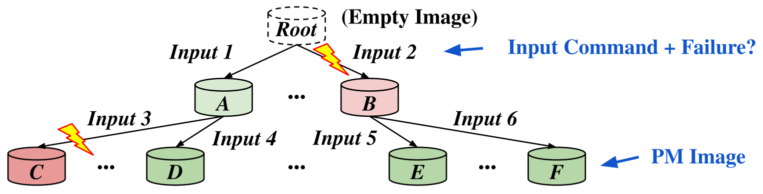


Figure 3C.12: Tree of PM images and input commands.

and crash images (step ❸). A crash image is generated by placing failures at each ordering point and additional failures at random locations (Section 3C.3.2); a normal image is the final outcome without any failure during the procedure. Then, the generated images go through a reduction procedure that eliminates any images that are identical to the previously generated ones (step ❹). The derandomization methods introduced in Section 3C.4.4 ensure that the same input test case always produces the same image. PMFuzz performs image reduction by looking up the image's hash value (SHA-256) in a dictionary that keeps the hash values of all prior images. Finally, both the newly generated commands and the resulting PM images will be reused as inputs in the next iteration of fuzzing (step ❺).

3C.4.6 Test Case Management

During fuzzing, test cases (input commands + a PM image) are generated recursively, by mutating prior test cases. PMFuzz efficiently manages the test cases by leveraging the dependencies among test cases. Figure 3C.12 demonstrates the dependencies, where each node is a PM image (the root is an empty image), and each edge represents the input command + failure location that are used to mutate the image. The image management method serves three main purposes. First, it makes the fuzzing procedure reproducible, as each test case and its resulting PM image can be tracked by the dependency. To reproduce a particular test case, the user can simply execute the input commands on top of its parent image. Second, test case tracking allows PMFuzz to incrementally generate test cases, by loading an existing PM image and executing a set of mutated input commands (the execution time is limited to 150 ms in this design), as Section 3C.4.5 has shown. Finally, the testing tool attached to PMFuzz (e.g., XFDetector [26] and Pmemcheck [128]) can also avoid executing redundant test cases. The testing tool only needs to execute a minimum set of test cases that cover new PM paths, without needing to start from prior test cases that contain the root image. For example, the test tool starts from test cases that contain the empty root image. Thus, to test the execution that produces image `D`, the testing tool only needs to execute `Input 4` on top of image `B`, as the execution that takes its predecessor (`Input 1` + `Root`) has been covered by the previous testing iterations.

3C.4.7 Optimization Strategies

In this section, we introduce three major optimizations in PMFuzz that improve the fuzzing efficiency.

(1) System Call Reduction. The fuzzing procedure takes multiple system calls when opening and closing PM images. The system call overhead can be further amplified when PMFuzz executes multiple fuzzing instances simultaneously. AFL++ comes with an optimization that creates multiple fuzzing instances using its fork server's copy-on-write mechanism (via `fork()`). It would significantly reduce the system call overhead of loading PM images if we can also copy-on-write persistent data on PM images. However, this method does not apply to PM images because they are memory-mapped (i.e., a file mapped to the program's virtual address space). To take advantage of the fork server in AFL++, when the PM program is opening a PM image, we first overload the `mmap()` function with our version that copies data from PM to a location on the heap of the program. Second, we use AFL++'s fork server to create multiple fuzzing instances, while carrying the persistent data that have been loaded from the PM image to the heap. Finally, before the PM program closes the image, we overload the `munmap()` function and save the updates back to the PM image as long as the execution has discovered new PM paths (based on the method in Section 3C.4.3). We validate this design to ensure that this optimization does not change the behavior by comparing the PM trace collected before and after applying this optimization (using Intel's Pin tool [177]).

(2) Test Case Storage. Fuzzing is a repeated process that generates a large number of test cases. Therefore, a PM device alone may not be sufficient to store all test cases. In our experiment, PMFuzz generated approximately 1.5 TB of data during a 4-hour period of fuzzing, primarily due to the PM images. Although PM images occupy a significant amount of space, we observe that the fuzzing procedure is periodical—PMFuzz takes a PM image as the input, spawns multiple fuzzer

Table 3C.1: System configuration.

CPU	Intel Xeon, 2.1 GHz, 20 cores
Memory	4×16 GB DDR4, 2666 MT/s 2×128 GB Intel DCPMM, Interleaved, App Direct Mode
SSD	2 TB, NVMe, PCI-E 3.0 ×4
OS	Ubuntu 18.04, Linux kernel v5.4
Environment	AFL++-2.63, LLVM-9, Clang-9, PMDK-1.8, Pin-3.13

instances, saves the generated images, and starts over again by taking the newly-generated PM images as inputs. In each iteration of fuzzing, only a small fraction of PM images will be taken as inputs. And, the generated PM images will not be used until the next iteration begins. Based on this observation, PMFuzz moves the generated test cases from the PM device to a hard drive (e.g., SSD) and compresses the generated PM images (using the LZ77 [190] algorithm). PMFuzz decompresses and moves an image back to PM, only when it is selected as the input. This optimization effectively reduces the storage requirement.

3C.5 Evaluation

3C.5.1 Methodology

System Configuration. We evaluate PMFuzz in a system with Intel's Cascade Lake processors and DC Persistent Memory Modules (DCPMMs), as listed in Table 3C.1. The PM devices (i.e., DCPMMs) are configured in the App Direct Mode and mounted with the `DAX` option to bypass OS indirections.

PM Programs. To evaluate PMFuzz, we choose PM programs (listed in Table 3C.3) built on top of Intel's PMDK (v1.8) [35] library, including simple key-value store structures [35] and real-world workloads [12, 14], similar to those tested by prior works [25, 26, 128, 129]. We use PMDK's `mapcli` [191] to drive the key-value stores, and use Preeny [189] to convert the socket-based communication interface of the databases to a command-line-based version.

Comparison Points. PMFuzz is developed on top of AFL++ (v2.63 [192]) with the integration of state-of-the-art fuzzing techniques, including LAF-Intel [193] and AFL-Sensitive [194]. Therefore, we take AFL++ as the main baseline fuzzer. To better demonstrate the impact of each PMFuzz

Table 3C.2: Comparison points.

	Input Fuzz	**Img Fuzz**	**PM Path Opt**	**Sys Opt**
PMFuzz (All Feat.)	Yes	Yes (Indirect)	Yes	Yes
PMFuzz w/o SysOpt	Yes	Yes (Indirect)	Yes	No
AFL++	Yes	No	No	No
AFL++ w/ SysOpt	Yes	No	No	Yes
AFL++ w/ ImgFuzz	No	Yes (Direct)	No	No

feature, we develop other alternative designs that are based on AFL++ and PMFuzz (listed in Table 3C.2). The details about the features are described below.

- **Input Fuzz** (Input Fuzzing) is a feature that mutates the input commands.
- **Img Fuzz** (PM Image Fuzzing) is a feature that mutates the PM image. The PM image is *indirectly* mutated using the program itself in the comparison point of PMFuzz but is *directly* mutated in AFL++ w/ ImgFuzz. As the baseline AFL++ does not support the mutation of both the image and the command input at the same time, we only enable image fuzzing in AFL++ w/ ImgFuzz.
- **PM Path Opt** (PM Path Optimization) is a feature that enables the targeted fuzzing on PM paths (introduced in Section 3C.4.3).
- **Sys Opt** (System-level Optimization) is a feature that reduces the system call and storage overhead (introduced in Section 3C.4.7).

Note that, in all comparison points, we enable the derandomization techniques (described in Section 3C.4.4) and use a list of basic commands and a PM image as the seed test case for fuzzing.

Detection Tool. PMFuzz is a test case generator that provides high-value test cases to the backend testing tools for PM programs. We leverage the most recent PM testing work XFDetector [26] as the testing tool attached to PMFuzz, which executes with PM programs and detects crash consistency and performance bugs. In addition, we use Intel's Pmemcheck [128] to detect synthetic bugs within the library (e.g., transaction, recovery, image creation, etc.).

Synthetic Bug Injection. To evaluate the effectiveness of test cases generated by PMFuzz, we place synthetic bugs in PM programs and the PDMK library, similar to the method taken by prior works [25, 26]. More specifically, we take the following approaches.

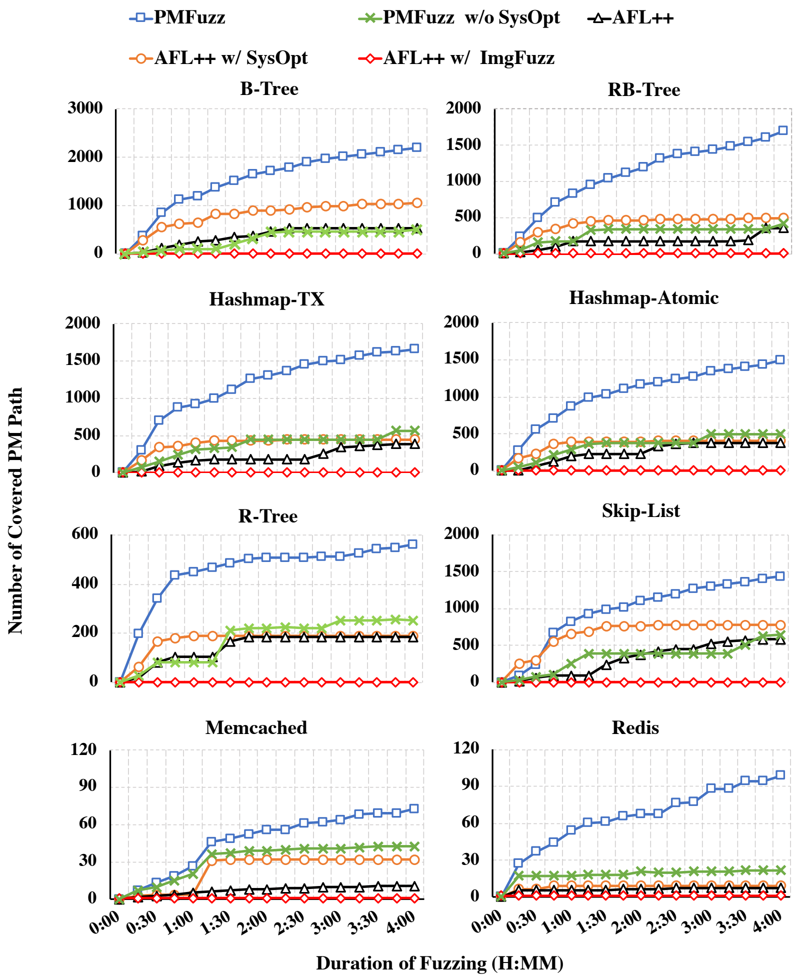


Figure 3C.13: PM path coverage.

- Remove/misplace writebacks (flushes) and fences to break the persistence requirement.
- Reorder PM writes that are originally ordered with write-backs and fences, to break the ordering requirement.
- Remove/misplace backup function calls to corrupt data in transaction-based programs.
- Place semantically incorrect code to cause incorrect recovery in programs based on low-level primitives, such as setting a wrong value to the commit variables.

3C.5.2 PM Path Coverage

Figure 3C.13 compares the number of unique PM paths covered by PMFuzz and the comparison points during 4-hour fuzzing. We summarize the results as the following points. (1) PMFuzz achieves

Table 3C.3: Tested PM programs, and synthetic bug detection.

	Program Name	**#Synthetic Bugs**	**#Covered by AFL++ SysOpt**	**#Covered by PMFuzz**
Simple KV-store	B-Tree	17	13	17
	RB-Tree	14	10	14
	R-Tree	16	12	16
	Skip-List	12	8	12
	Hashmap-TX	21	16	21
	Hashmap-Atomic	14	10	14
Real-world Workloads	Memcached	17	14	17
	Redis	14	9	14

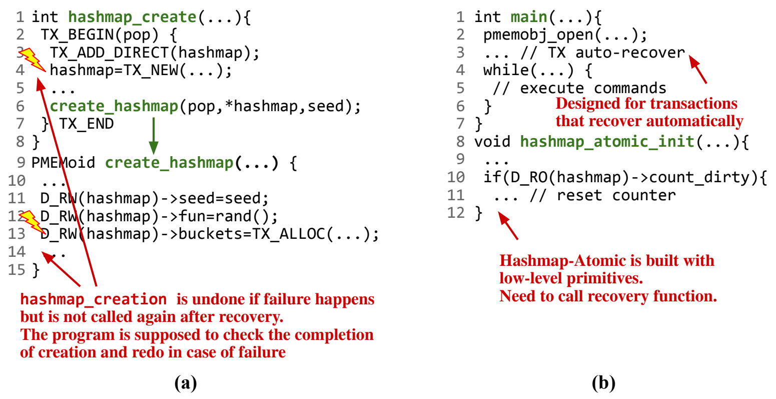
a significant increase in PM path coverage over AFL++ (Geo-mean 4.6×) because it efficiently mutates PM images, performs a targeted fuzzing on PM path, and consumes a low system overhead. (2) The PM path coverage is significantly lower without our system optimizations (PMFuzz w/o SysOpt), demonstrating that the system-level optimizations are essential to fuzzing PM programs. (3) AFL++ with system optimizations (AFL++ w/ SysOpt) outperforms AFL++ (Geo-mean 1.4×), but still cannot provide comparable coverage to PMFuzz. (4) AFL++ with PM image fuzzing (AFL++ w/ ImgFuzz) has poor coverage progress due to the large search space within PM images. Finally, the two databases, Memcached and Redis have fewer PM paths as compared to other key-value store structures. The primary reason is that only a relatively small fraction of code manages PM. Additionally, it takes much longer to execute them due to their higher complexity.

3C.5.3 Synthetic Bug Detection

Table 3C.3 lists the number of synthetic bugs tested and detected by PMFuzz. We compare PMFuzz with AFL++ w/ SysOpt in this experiment, as this configuration performs the best among the non-PMFuzz comparison points. We observe that PMFuzz generates test cases that detect *all* synthetic bugs, 1.4× over AFL++ w/ SysOpt, due to PMFuzz's effective PM image generation (both normal and crash images) and the focus on PM paths. Worth pointing out that the software development for PM is currently in an early stage. Therefore, the existing workloads are relatively simple. We expect that PMFuzz will show a more prominent advantage over conventional fuzzers while testing future real-world PM programs.

3C.5.4 New Real-world Bugs Found by PMFuzz

Despite the fact that prior works [25,26,128] have intensively tested PM programs listed in Table 3C.3, test cases generated by PMFuzz help detect new real-world bugs.



```
1 int hashmap_create(...){
2  TX_BEGIN(pop) {
3   TX_ADD_DIRECT(hashmap);
4   hashmap=TX_NEW(...);
5   ...
6   create_hashmap(pop,*hashmap,seed);
7  } TX_END
8 }
9 PMEMoid create_hashmap(...) {
10  ...
11  D_RW(hashmap)->seed=seed;
12  D_RW(hashmap)->fun=rand();
13  D_RW(hashmap)->buckets=TX_ALLOC(...);
14  ...
15 }
```

```
1 int main(...){
2  pmemobj_open(...);
3  ... // TX auto-recover
4  while(...) {
5   // execute commands
6  }
7 }
8 void hashmap_atomic_init(...){
9  ...
10  if(D_RO(hashmap)->count_dirty){
11   ... // reset counter
12 }
```

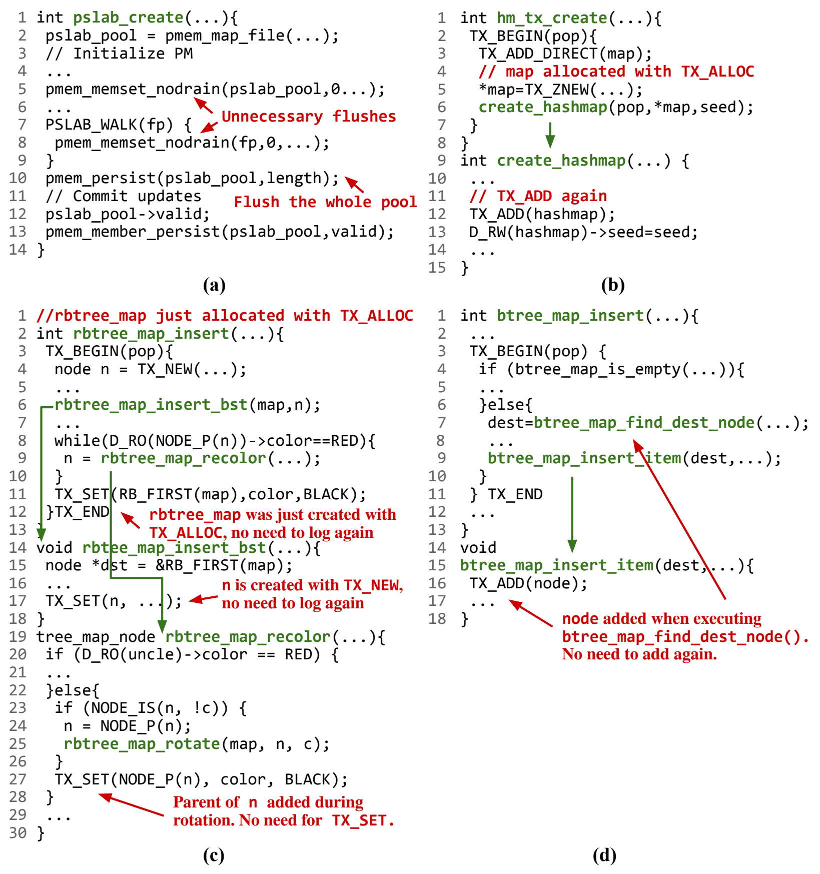
hashmap_creation is undone if failure happens
but is not called again after recovery.
The program is supposed to check the completion
of creation and redo in case of failure
Designed for transactions
that recover automatically
Hashmap-Atomic is built with
low-level primitives.
Need to call recovery function.
(a)
(b)

Figure 3C.14: New crash consistency bugs found by PMFuzz: (a) Bug 1 and (b) Bug 6.

New Crash Consistency Bugs. **Bug 1-5:** Figure 3C.14a is a simplified code snippet from Hashmap-TX (`hashmap_tx.c:402`), where `create_hashmap` uses a transaction (line 2-7) to allocate space and initialize the hash table. PMFuzz created two crash images before and within the allocation. When taking the crash images for the next fuzzing iteration, both of them report a segmentation fault when the program attempts to dereference the pointer to `hashmap`. We found that `hashmap_create` is called when starting with an empty PM image. In case the procedure fails, the whole creation procedure is undone by the transaction, leaving `hashmap` a `NULL` pointer. However, because the program does not call `hashmap_create` again afterward, the following execution assumes a fully initialized hash table. Other 4 transactional workloads, including B-Tree, RB-Tree, R-Tree, and Skip-List also have similar bugs during initialization. Although the prior failure-aware testing tool XFDetector [26] can detect this type of bugs with a simple test case of an empty PM image, due to the programmer's effort in understanding and annotating the source code, XFDetector did not take the buggy code region into consideration.

Bug 6: Figure 3C.14c shows two functions: `main()` is a driver program for PMDK's key-value store, Mapcli (`mapcli:205`). The other function, `hashmap_atomic_init()`, is a procedure in Hashmap-Atomic (`hashmap_atomic.c:452`). This code snippet has a crash consistency bug as the `main()` function assumes all key-value store structures can automatically recover using transactions, but overlooks the low-level-primitive-based Hashmap-Atomic. Detecting this bug requires a test case that has `counter_dirty=true` (line 10), which is not easy to reach without a PM-specific test case generator.

New Performance Bugs. **Bug 7:** Figure 3C.15a is a code snippet from Memcached (`pslab.c:317`) that creates a new `pslab_pool`. It starts with setting up a few metadata entries, and then flushes the whole pool. Finally, it sets a `valid` bit (surrounded with ordering points) to



```
1  int pslab_create(...){
2   pslab_pool = pmem_map_file(...);
3   // Initialize PM
4   ...
5   pmem_memset_nodrain(pslab_pool,0...);
6   ...
7   PSLAB_WALK(fp) {
8    pmem_memset_nodrain(fp,0,...);
9   }
10  pmem_persist(pslab_pool,length);
11  // Commit updates
12  pslab_pool->valid;
13  pmem_member_persist(pslab_pool,valid);
14 }
```

(a)

```
1  int hm_tx_create(...){
2   TX_BEGIN(pop){
3    TX_ADD_DIRECT(map);
4    // map allocated with TX_ALLOC
5    *map=TX_ZNEW(...);
6    create_hashmap(pop,*map,seed);
7   }
8  }
9  int create_hashmap(...) {
10  ...
11  // TX_ADD again
12  TX_ADD(hashmap);
13  D_RW(hashmap)->seed=seed;
14  ...
15 }
```

(b)

```
1  //rbtree_map just allocated with TX_ALLOC
2  int rbtree_map_insert(...){
3   TX_BEGIN(pop){
4    node n = TX_NEW(...);
5    ...
6    rbtree_map_insert_bst(map,n);
7    ...
8    while(D_RO(NODE_P(n))->color==RED){
9     n = rbtree_map_recolor(...);
10   }
11   TX_SET(RB_FIRST(map),color,BLACK);
12  }TX_END
13
14 void rbtree_map_insert_bst(...){
15  node *dst = &RB_FIRST(map);
16  ...
17  TX_SET(n, ...);
18 }
19 tree_map_node rbtree_map_recolor(...){
20  if (D_RO(uncle)->color == RED) {
21  ...
22  }else{
23   if (NODE_IS(n, !c)) {
24    n = NODE_P(n);
25    rbtree_map_rotate(map, n, c);
26   }
27   TX_SET(NODE_P(n), color, BLACK);
28  }
29  ...
30 }
```

(c)

```
1  int btree_map_insert(...){
2   ...
3   TX_BEGIN(pop) {
4    if (btree_map_is_empty(...)){
5    ...
6    }else{
7     dest=btree_map_find_dest_node(...);
8     ...
9     btree_map_insert_item(dest,...);
10   }
11  } TX_END
12  ...
13 }
14 void
15 btree_map_insert_item(dest,...){
16  TX_ADD(node);
17  ...
18 }
```

(d)

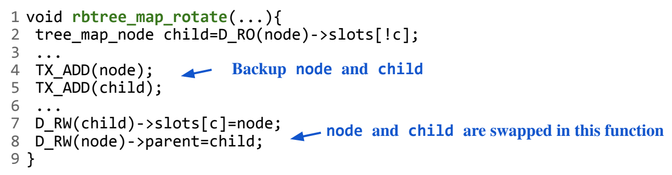
Unnecessary flushes
Flush the whole pool
rbtree_map was just created with TX_ALLOC, no need to log again
n is created with TX_NEW, no need to log again
Parent of n added during rotation. No need for TX_SET.
node added when executing btree_map_find_dest_node(). No need to add again.

Figure 3C.15: New performance bugs found by PMFuzz: (a) Bug 7, (b) Bug 8, (c) Bug 9–11, and (d) Bug 12.

commit the creation (line 12). There are two redundant flushes (line 5 and 8) to the metadata as line 10 flushes the whole `pslab_pool`.

Bug 8: Figure 3C.15b is a code snippet from Hashmap-TX that performs insertion (`hashmap_tx.c:90`). Line 12 calls a redundant `TX_ADD()` to back up a node that was previous allocated by `TX_ZNEW()` (line 5) which has logged this object.

Bug 9–11: Figure 3C.15c is a code snippet from RB-Tree showing the procedure of an insertion function that contains three performance bugs (`rbtree_map.c:215`). **Bug 9** is at line 17 that uses `TX_SET()` to update the transaction-allocated node `n`, which introduces a redundant log operation. **Bug 10** is at line 11 that logs `RB_FIRST(map)`, which is the first entry in the tree, before performing the update. However, if the tree was just transaction-allocated (comment at line 1), it is unnecessary to log it again. **Bug 11** is at line 27 that uses `TX_SET()` to update the `parent` of node `n`, which has been backed up if `rbtree_map_recolor()` executes `rbtree_map_rotate()` first. These performance bugs can be detected by prior testing tools but require a specific test case to trigger. In particular, Bug 9 can be detected only when testing a newly allocated tree, and Bug 11 requires the if-condition



```
1 void rbtree_map_rotate(...){
2  tree_map_node child=D_RO(node)->slots[!c];
3  ...
4  TX_ADD(node);
5  TX_ADD(child);
6  ...
7  D_RW(child)->slots[c]=node;
8  D_RW(node)->parent=child;
9 }
```

Figure 3C.16: An example from RB-Tree that demonstrates the trade-off between programmability and performance.

at line 20 to be false but line 23 to be true.

Bug 12: B-Tree has a performance bug as shown in Figure 3C.15d (`btree_map.c:276`). `btree_map_insert()` first finds the destination using `btree_map_find_dest_node()` and then inserts the node using `btree_map_insert_item()`. `TX_ADD()` at line 16 is unnecessary because `node` has been added when finding the destination (performs modification if tree-split is needed).

Efficiency of Test Case Generation. PMFuzz is also efficient in generating test cases that detect those bugs. To cover Bug 1–5, 7, and 8, PMFuzz only took 2 seconds of wall-clock time—as soon as the first batch of test cases was generated, since those bugs are located in the initialization step. For the rest of the bugs, Bug 9 and 10 are detected by the same case that took 91 seconds to generate; Bug 6, 11, and 12 took 37, 77, and 88 seconds, respectively. The fuzzing time was longer as covering those bugs requires relatively more complex program paths.

3C.6 Discussion

In this section, we discuss the trade-offs between programmability and performance, and then the potentials for extending PMFuzz to accommodate other types of PM software systems.

Performance Bug Trade-offs. In the PMDK library, a redundant `TX_ADD()` does *not* create additional logs. All logged locations are kept track of using a *range tree*. Before creating a new log entry, the library looks up the location in the *range tree* to make sure it has not been logged before. With this mechanism, it is safe to call `TX_ADD()` without checking conditions, such as whether the object has been backed up or allocated with a transactional interface. Nonetheless, the unnecessary range tree lookup can still lead to performance penalties (e.g., Bug 9–12). Therefore, we expect highly-optimized PM programs to avoid these redundant calls to transactional functions.

On the other hand, it is sometimes hard to completely remove performance bugs. Figure 3C.16 shows an example from RB-Tree (`rbtree_map.c:189`), where `rbtree_map_rotate()` swaps `node` with its `child` (line 7 and 8). If this function is called multiple times, i.e., keep rotating until the tree is balanced, the two `TX_ADD()` calls (line 3 and 4) can apply to objects that have already been logged. However, it is hard to tell whether or not a node has been logged as the rotation depends on the value of each node. Instead, it is easier to implement the rotation procedure by logging both nodes in the beginning to avoid any crash consistency issues. Therefore, we do not treat this type of issue as a performance bug.

Integration with PM Kernel Modules. There have been works that develop PM-optimized file systems for other programs to manage persistent data [8–11, 161, 195]. These file systems are implemented as kernel modules but different from conventional file systems, they customize the persistent data, much like the user-space PM programs. Thus, it is hard to directly mutate their PM images. PMFuzz runs in the user-space as it is built upon AFL++ [182]. Nonetheless, it is possible to convert kernel-mode file systems into user-space programs, using libraries such as Linux Kernel Library (LKL) [196], or execute them on a virtual machine [181, 184]. This way, PMFuzz can be integrated into such frameworks to generate test cases for kernel-mode, PM-optimized file systems. We leave this direction as a future work.

Multithreading. PM programs may run in multithreaded mode for better throughput. PMFuzz is built on top of AFL++ which is thread-safe. However, multithreading introduces randomness due to various conditions of thread interleavings. As randomness prevents the fuzzer from converging to good coverage, it is not recommended to run PMFuzz with multithreading-enabled programs. On the other hand, recent works have pointed out potential persistency issues with multithreaded execution [197, 198]. PMFuzz's targeted fuzzing on PM operations can generate high-value test cases for such scenarios, with an extended focus on PM-related multithread synchronization primitives. We leave test case generation for multithreaded PM programs as a future work.

Chapter 4

Secured and Efficient Hardware for Persistent Memory Systems

Chapter 4A

Crash Consistency in Encrypted Persistent Memory Systems

4A.1 Introduction

Chapter 3 focuses on challenge in ensuring crash consistency of PM software systems, i.e., persistent data can be recovered to a consistent state in event of a system failure. An orthogonal challenge in designing PM systems concerns with the security of persistent data. Data in PM remains persistent across system failures. Therefore, vulnerable to malicious attackers who have physical access to the devices [50, 54, 55]. Encryption is an effective solution to protect PM data from the attackers. In an encrypted PM system, every read access to memory needs to pay an additional penalty for decrypting data in the memory controller. A common memory encryption technique referred to as the *counter-mode encryption* [17], has been adopted for PM to reduce the high overhead of decryption latency during a memory read access [48–51, 54, 199]. The counter-mode encryption technique associates each cache line of data with a counter such that the cache line is encrypted with a bit string generated with the associated counter. The same bit string is used to decrypt the cache line on subsequent read accesses to memory. The counter-mode encryption hides the decryption latency by generating the bit string for decryption using the counter buffered on-chip in a counter cache, while the data is still being fetched from memory [48, 52, 199]. In this work, we show that, even though the counter-mode encryption technique hides the decryption latency for the critical memory read accesses, it *does not* readily extend to PM systems that require data to be recoverable in a consistent state across system failures. As the counters and data are located in different addresses, every write to PM generates two write requests: one for the encrypted data and the other for the counter. These two writes to PM (for the encrypted data and counter) have to be performed *atomically* to ensure that persistent data in memory can be decrypted across system failures. For example, if the system fails after the encrypted data reaches PM, but before its counter has been persisted, the memory controller will try to decrypt that data using a *stale* counter value upon recovery and will *fail* to recover the original data. We refer to the constraint of counter and encrypted data being updated in PM atomically as *counter-atomicity* and argue that the encrypted PM systems need to provide support for *counter-atomicity* to ensure crash consistency in PM systems.

Ensuring *counter-atomicity* is challenging as existing systems cannot atomically write two accesses to memory. One solution to this problem is to extend the cache line to co-locate data and its counter in the same cache line and then, use a wider memory bus to write back the extended cache line *atomically* using just *one* write access. Unfortunately, this design is impractical as widening the memory bus to accommodate an extra counter requires extra pins in the memory interface, exacerbating the problem of limited memory bandwidth [200–202]. The *goal* of this work is to design an *PM system that enforces counter-atomicity at a low cost and a low overhead.*

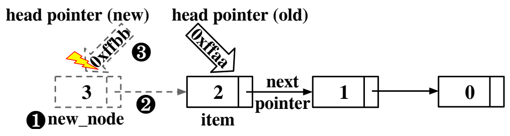
We propose a simple hardware mechanism to enforce *counter-atomicity* in a PM system. A special write queue in the memory controller ensures that either *both* data and its counter of a write access have been persisted or *neither* of them has been persisted. Unfortunately, ensuring *counter-atomicity* for every write access to memory potentially makes each pair of data and counter write sequential. It results in a significant performance degradation, restricting the optimizations through reordering, buffering, and coalescing of writes in the memory controller. However, we observe that it is still possible to decrypt and recover data consistently even when all writes are *not* enforced to be *counter-atomic*. Our *key insight* is that only a small subset of writes to PM need counter-atomicity to be strictly enforced in order to maintain recoverability of the persistent data. For example, the insertion of a new node to a persistent linked list in PM consists of two sets of writes: one set of writes creates a new node with valid data and the other updates the head pointer of the list to point to the new node. The writes related to the creation of the new node have to reach PM before the write to the pointer to ensure the recoverability of the list in a consistent state. Therefore, the writes to the new node *do not* affect the recoverability of the linked list until the write to the pointer reaches PM. The writes to the node and the corresponding counter updates can be coalesced, buffered or reordered as long as they are performed before the write to the pointer, while the write to the pointer itself requires *strict* counter-atomicity. This observation also extends to the common crash consistency mechanisms, such as undo/redo logging, shadow copying, etc., which rely on *versioning* of data updates to ensure crash consistency. For example, the logging mechanism maintains one version of data in the log and another version in the original data structure. It ensures that at a given point in time, only *one* of the versions is modified so that the other version can be used to recover data if there is a crash during the update. As the version of data being modified plays no role in recovery, strictly enforcing *counter-atomicity* for those writes is not necessary. Therefore, we propose that PM writes that *do not* manipulate the recoverable state provide a window during the program execution when the data and counter writes can be reordered to significantly improve performance. We refer to this design as *selective counter-atomicity* (details in Section 4A.4). We propose necessary software interface and hardware support to *selectively* enforce *counter-atomicity* in a PM system (details in Section 4A.5).

To summarize, the contributions of this work are:

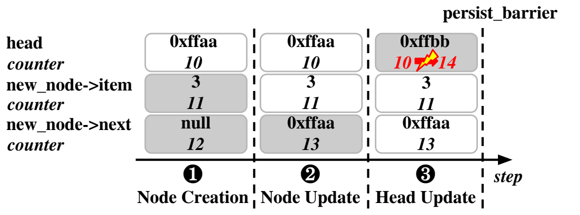
- We show that the commonly used counter-mode encryption *does not* extend to the PM systems that require data to be recoverable in a consistent state across system failures. This is the *first work* to introduce the requirement of *counter-atomicity* that ensures both data and the associated counter have to be persisted *atomically* in order to guarantee crash consistency in an encrypted PM system.

- We introduce *selective counter-atomicity* by demonstrating that it is not required to enforce counter-atomicity for *all writes*. We observe that the common PM crash consistency mechanisms rely on versioning of data. Data updates to one of the versions do not immediately affect the consistent state, and therefore, it is possible to *selectively* enforce counter-atomicity for *only* the writes that manipulate the recoverable state. The rest of the data and counter writes can be coalesced, buffered, and reordered to improve performance.

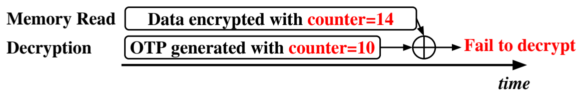
- Our evaluation demonstrates that *selective counter-atomicity* improves performance by 6/11/22/40% over enforcing counter-atomicity for *all writes* in a system with 1/2/4/8 cores, and it performs within 5% of an ideal design that does not have any overhead in enforcing counter-atomicity in our evaluated system configurations.



(a) The steps in adding a new node to a persistent linked list.



(b) The timeline of the data and counter update at each step. The shaded boxes represent the updated values in each step.



(c) Recovery fails due to inconsistent data and counter values of the `head` pointer.

Figure 4A.1: An example of inconsistency while adding a node to a persistent linked list.

4A.2 Crash Consistency for Encrypted PM Systems

Encrypting PM is highly important for protecting data, as attackers who have physical access to a PM module can access information stored in the persistent memory [50, 54]. Section 2.3.2 shows that directly applying the existing crash consistency mechanism to an encrypted PM system *does not* guarantee a consistent recovery of data in case of a power failure or system crash. In this section, we further demonstrate the consequences of the inconsistency between data and counter using an example.

Figure 4A.1a demonstrates a linked list where each node contains an `item` and a `next` pointer to the consecutive node, and the `head` pointer points to the most recently added node. Adding a new node involves three steps as shown in the Figure: The first step creates a new node (step ❶). The step ❷ updates the `next` pointer of the new node, so that it is inserted in front of the current `head` of the list. Finally, in step ❸, the head pointer is updated so that it points to the new node. When the linked list is encrypted, each update in the linked list becomes associated with a counter update. Figure 4A.1b shows the plaintext data and counter values at each step, where the shaded boxes represent the updated values. In the beginning, the `head` points to the next node and its associated counter value is "`10`". At step ❶ and ❷, the new node is updated with its item and the new pointer value and the associated counters are also updated with new values. At step ❸, the head pointer is updated to point to the new node and the latest counter value for the `head` becomes "`14`". This means that the value of the `head` pointer gets encrypted with the latest counter

"`14`" before it is persisted to memory. However, if a failure happens before the new counter value "`14`" gets persisted to PM, the values of the `head` pointer and its associated counter in PM become out-of-sync. During the recovery, the decryption engine will try to decrypt that the `head` pointer with the stale counter ("`10`"), making decryption unsuccessful (shown in Figure 4A.1c). Potentially, the value of the `head` pointer can become a random number after the incorrect decryption, causing the program to mistakenly access a random location in memory. To support crash consistency in an encrypted PM, we argue that it is required to enforce an *atomic* behavior of the counter and data writes, which we refer to as *counter-atomicity.*

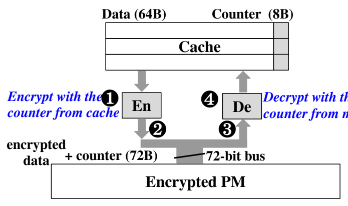
This chapter targets two goals. First, demonstrate that the encrypted data and associated counter need to be *atomic* in order to support crash consistency in encrypted PM systems (Section 4A.3.1). Second, discuss the challenges of the possible solutions to provide this *counter-atomicity* to enforce an atomic behavior of the counter and data writes (Section 4A.3.2). Third, propose an efficient hardware-software design to enforce counter-atomicity based on the key observation that not all writes need to be *counter-atomic* (Section 4A.4).

4A.3 Counter-Atomicity

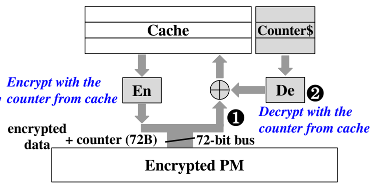
The key to maintaining crash consistency in an encrypted PM system using counter-mode encryption technique is to guarantee that data and the associated counter are persisted in an *atomic* manner. We refer to this requirement as *counter-atomicity* for crash consistency in an encrypted PM. In this section, first we define the requirement of counter-atomicity. Then, we discuss the trade-offs in the designs that meet the requirement of *counter-atomicity.*

4A.3.1 Requirement

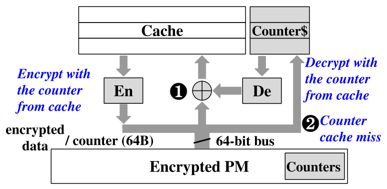
A *counter-atomic write* operation needs to guarantee that either both data and its counter associated with the write access have persisted (the *counter-atomic write* is *complete*) or neither data nor its counter has persisted (the *counter-atomic write* is *incomplete*) in case of a system crash. This requirement prevents a mismatch in version for data and counter values in a *counter-atomic write.*



(a) The co-located data and counter design with a wider bus.

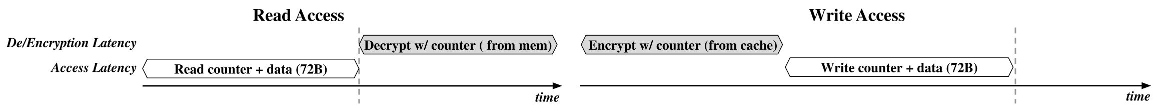


(b) The co-located data and counter design with a wider bus and a counter cache.

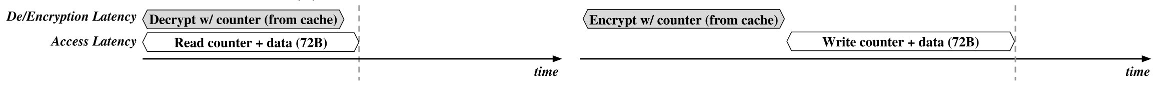


(c) The separate data and counter design with the existing bus and a counter cache.

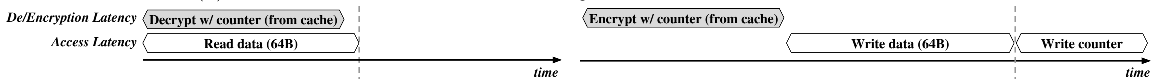
Figure 4A.2: Different counter-atomic designs.


Read Access
Write Access
De/Encryption Latency
Access Latency
Read counter + data (72B)
Decrypt w/ counter (from mem)
Encrypt w/ counter (from cache)
Write counter + data (72B)
time
time

(a) The co-located data and counter design with a wider bus.


De/Encryption Latency
Access Latency
Decrypt w/ counter (from cache)
Read counter + data (72B)
Encrypt w/ counter (from cache)
Write counter + data (72B)
time
time

(b) The co-located data and counter design with a wider bus and a counter cache.



(c) The separate data and counter design with the existing bus and a counter cache.

Figure 4A.3: Timeline of read and write accesses with three different design shown in Figure 4A.2.

4A.3.2 Enforcing Counter-atomicity

In this section, we describe the challenges in enforcing *counter-atomicity*, propose simple hardware designs to solve the challenges, and discuss the trade-offs in each design.

Challenge 1: How to ensure data and counter reach PM at the same time?

In today's systems, there is no guarantee that the separate counter and data writes will reach the PM at the same time. If a failure happens in the middle of the counter and data writes, that data cannot be decrypted due to the mismatch in the versions of data and counter. A naïve solution is to write both data and the associated counter together with *one memory access* by co-locating them in the same access. To accommodate the extra counter, such a design requires *(i)* increasing the size of the cache line in the last-level cache (LLC), and *(ii)* increasing the width of the memory bus. As every cache line of data needs an 8B counter in the counter-mode encryption (as shown in prior works [51,52,199]), a typical cache line size will increase from 64B to 72B and the memory width will increase from 64-bit to 72-bit. We refer to this design as the *co-located data and counter design with a wider bus*. Figure 4A.2a shows the high-level organization of this design. During a write access, the memory controller first encrypts the data (step ❶) and then writes the encrypted data and its counter simultaneously to PM using the wider bus (step ❷). However, this design is not efficient as it serializes the read access and the decryption process. The memory controller first needs to fetch both data and its counter from memory (step ❸) and only then it can decrypt that data using the co-located counter fetched from PM (step ❹). Such a serialized design violates the main benefit of the counter-mode encryption technique. Figure 4A.3a shows the timeline of the serialized read access and write access of this design. However, it is possible to mitigate the decryption overhead by adding a counter cache, as shown in Figure 4A.2b. The cached counters enable overlapping the decryption process with the read access. While the missed cache line is being fetched from the PM (step ❶), the memory controller starts generating the OTP using the counter from the counter cache (step ❷). However, in this design, if the requested counter is not in the counter cache, the memory access results in a counter cache miss and the memory controller fetches the entire cache line again, as the data and counter are co-located and the access granularity is 72B. We refer to this design as the *co-located data and counter design with a wider bus and a counter cache.* Figure 4A.3b shows the timeline of this improved design, where the read latency overlaps the decryption latency if the counter cache lookup results in a hit.

Trade-offs. The benefit of co-locating the data and counter in one memory access is that this design eliminates any chance of having the data and counter values out-of-sync in PM and therefore, always guarantees that the writes will be *counter-atomic.* However, as the cache line size increases to 72B (64B data + 8B counter), this design requires increasing the memory bus width from 64-bit to 72-bit. As a result, the counter writeback requires extra pins and wires in the memory bus, exacerbating the problem of limited memory bandwidth [200–202]. We believe that widening the memory bus is not practical and study alternative designs that can enforce *counter-atomicity* with the existing memory interface.

Challenge 2: How to enforce *counter-atomicity* without changing the memory interface?

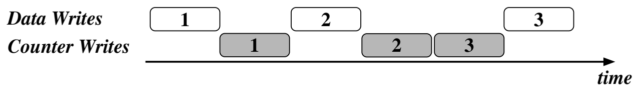
The major drawback in the two aforementioned designs (Figure 4A.2a and 4A.2b) is that they require an expensive and impractical change in the memory interface. Therefore, a more practical design is to write back data and counters using separate write requests, but provide some hardware support to ensure that the write accesses are *counter-atomic*: a memory write request is marked as *complete* only when both data and counter have become *persistent.* We propose a simple hardware support in the memory controller that tracks data and the associated counter in the *write queue* and ensures that the write access is blocked until both the data and counter become persistent. We discuss the details of the implementation of this design in Section 4A.5.

Figure 4A.2c shows the high-level organization of this design. As the data and counter are written separately with two different write accesses, they are not co-located in PM. Instead, the counters are stored in a separate address space. For the same reason, the memory bus remains unchanged (64-bit). The read access is similar to the previous designs where the read access and decryption happen in parallel (step ❶). When the read access misses the counter in the counter cache, the memory controller fetches a whole cache line of counters from memory (step ❷). Figure 4A.3c shows the timeline of this design. The latency to *complete* a write request becomes higher as a single write request now consists of two accesses (one for the data cache line and one for the counter cache line).

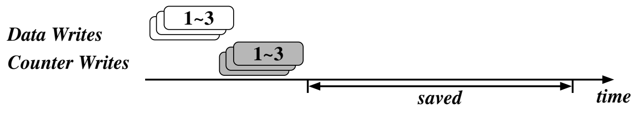
Trade-offs. This simple implementation not only mitigates the overhead of the serialized read access and decryption latency, but also ensures *counter-atomicity* without changing the memory interface. However, this mechanism leads to performance degradation as every write access becomes counter-atomic, blocking other dependent writeback requests if either the data or counter write request has not yet been persisted and therefore, can potentially serialize all write accesses. We refer to this design that always writes back data and counter in a *counter-atomic* manner as the *full counter-atomicity* design. In Section 4A.4, we propose an optimization where only a subset of the write accesses needs to maintain *counter-atomicity*, but still guarantees that the system remains *crash consistent*.

4A.4 Selective Counter-Atomicity

In this section, first, we discuss the high overhead of enforcing *full counter-atomicity* (Section 4A.4.1). Then we propose to mitigate its overhead by *selectively* enforcing *counter-atomicity* to a small subset of writes without affecting the recoverability of programs in a consistent state based on the observation that not all writes equally affect consistent data recovery (Section 4A.4.2). We refer to this design as *selective counter-atomicity* and provide necessary interface and primitives to leverage it in different persistent applications (Section 4A.4.3).



(a) Full counter-atomicity.



(b) Without counter-atomicity.

Figure 4A.4: The timeline of write accesses in a *full counter-atomicity* design vs. an *ideal* design that does not enforce *counter-atomicity.*

4A.4.1 The Overhead of *Full Counter-Atomicity*

Enforcing *counter-atomicity* is necessary to make sure that data in PM is consistent across system failures. In this design, a write access is *complete* only when both the data and associated counter are persistent. Strictly enforcing *counter-atomicity* for *all* writes to PM leads to high performance overhead in two ways. First, every write access has to initiate a corresponding counter write access. It doubles the amount of write traffic as our design writes back data and counter at a cache line granularity with two separate write accesses. Though a write access needs to update *only* one counter for the whole cache line of data, in this design, the counter is updated at a cache line granularity, which unnecessarily increases the write traffic. In multi-core systems, this extra contention between data and counter writeback becomes more prominent. Second, the write access blocks dependent writes until both the data and counter write accesses are complete. Figure 4A.4a shows the timeline of a sequence of updates in a *full counter-atomicity* design, where the white and shaded boxes represent the write accesses for the data and counters, respectively. The figure demonstrates the worst-case scenario where each write access is dependent on the prior one. The second write access has to wait until the first one completes and the third write access has to wait until the second one completes. In comparison to this design, a write access does not wait for its counter write to complete to make forward progress in an *ideal* design that does not require *counter-atomicity.* As a result, the write accesses can be reordered, coalesced, and written back in parallel, as shown in Figure 4A.4b. In the next section, we propose a design that reduces the overhead of the *full counter-atomicity* design leveraging the key insight that *not all* writes need to be *counter-atomic* to ensure consistent data recovery in a PM system.

4A.4.2 Not All Writes Equally Affect Recoverability

We make an observation that not all write accesses equally affect the recoverability of data in persistent applications. Persistent applications usually build upon some transactional interface to provide crash consistency across system failures. For example, undo logging, redo logging, shadow logging, journaling, etc. provide a guarantee that data can be recovered in a consistent state even if there is a failure during an update [36,125,126,161,173,203]. All these mechanisms guarantee crash consistency by maintaining two versions of data. For example, the logging mechanism maintains one version in the log and another version in the original data structure. Therefore, the program ensures that only one of the versions is being actively modified at a given point in time. While one of the versions of data is being modified, the other unmodified version is used for recovering the consistent state, if there is any crash. As the version of data being modified plays no role in recovery, it is not required to *strictly* enforce *counter-atomicity* for the writes to that version of data. However, these updates to the modified version need to be persisted in PM *before* the old version becomes stale and the modified version becomes the updated new consistent version. Therefore, it is possible to guarantee a consistent recovery even without strictly enforcing *counter atomicity* for all writes as long as these updates are persisted in PM *before* they start affecting the recoverability of the system.

Key Insight. Based on the observation that a subset of writes to PM *does not* immediately affect the crash consistent recovery of the underlying data structure, but instead affects consistent recovery only *after a certain future point in program execution*, our key insight is to relax the requirement of *counter-atomicity* during these windows of program execution. Therefore, instead of enforcing *full counter-atomicity* for all writes, we allow coalescing, buffering, and reordering of both the data and counter writes during these windows of program execution, as long as they are drained to PM at the end of the window. Based on this key insight, we propose the *selective counter-atomicity* design that only enforces *counter-atomicity* for a subset of write accesses to provide better performance without affecting the crash consistency guarantee.

***Selective Counter-Atomicity* in a Transaction.** In this section, we show how *selective counter-atomicity* can be applied to improve the performance of a transaction implemented using undo-logging. Each transaction consists of three stages as shown in prior works [1,35,125]:

1. **Prepare.** A log entry is created to back up the data being modified.
2. **Mutate.** The data structure is modified in-place. As a consistent state of the data is available in the backup created in the *prepare* stage, this in-place modification does not affect the recoverability of data.

Table 4A.1: The consistency states affecting *counter-atomicity* in different stages of a transaction with undo-logging.

Stage	**Backup**	**Data**	**Counter-Atomicity**
Prepare	✘ Inconsistent	✔ Consistent	✘ Unnecessary
Mutate	✔ Consistent	✘ Inconsistent	✘ Unnecessary
Commit	? Unknown	? Unknown	✔ Necessary

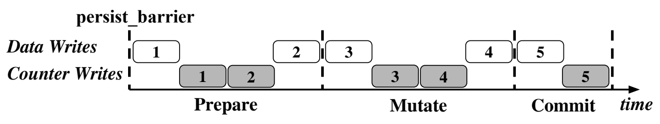
3. **Commit.** Once data modification is finished, the transaction is committed by invalidating the backup log entry created in the *prepare* stage and marking the new modified state as the current consistent state.

We summarize these stages in Table 4A.1 and show when *counter-atomicity* is necessary for each stage. During the *prepare* stage, the backup copy of the data in the log is being modified and therefore, cannot be used for consistent recovery, while the original data is unmodified and used to recover data in a consistent state. These writes to PM in the *prepare* stage do not immediately affect the recoverability and do not need to be strictly *counter-atomic*. Similarly, during the *mutate* stage, the backup copy in the log is consistent and can be used for consistent recovery, while the original data is being modified and thus, is not used for recovery. Therefore, the writes in the mutate stage *do not immediately* affect the recoverability and *do not* need *strict* counter-atomicity. On the other hand, the write in the *commit* stage atomically invalidates the backup log entry. The consistent version of data remains in the log entry until the *commit* stage, which switches the consistent state from the log to the modified data in the original place. As the write in this stage *immediately* affects the recovery of data in a consistent state by marking which version of data to use during the recovery procedure, the writes in this stage need to be strictly *counter-atomic*.

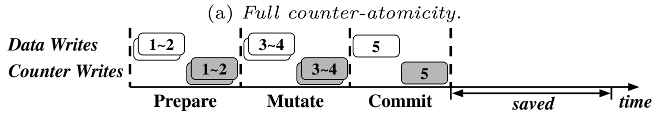
Figure 4A.5 shows the timeline of writes in different stages of a transaction with both *selective counter-atomicity* and *full counter-atomicity* (detailed performance analysis in Section 4A.6.3). Figure 4A.5a shows the case where enforcing *full counter-atomicity* serializes the writes in each stage. On the other hand, *selective counter-atomicity* allows the counter and data write accesses in the *prepare* and *mutate* stages to be reordered such that the write accesses can be performed in parallel (as shown in Figure 4A.5b). However, this figure shows that *counter-atomicity* must be enforced for the write accesses in the *commit* stage.

4A.4.3 Definition and Primitives

A *selective counter-atomicity* design has two requirements, *(i)* strictly enforce *counter-atomicity* only for those updates that immediately affect the recoverability of data in a consistent state, and *(ii)*



(a) *Full counter-atomicity.*



(b) *Selective counter-atomicity.*

Figure 4A.5: Timeline showing three stages of a transaction with undo-logging under *full counter-atomicity* and *selective counter atomicity.*

allow coalescing, buffering and reordering of all other data and counter writes during the program execution until they affect the recoverability of data. To this end, we propose two new primitives to extend Intel's persistency support [3]. We expose two counter-related primitives to the high-level program in order to let the programmers leverage the benefits of *selective counter-atomicity*:

`CounterAtomic` variables. Any variable that immediately affects the recoverability of the underlying data structure must be defined as `CounterAtomic`. The hardware is responsible to ensure that any update to this variable will write back the encrypted value and the associated counter *atomically*. For example, the `head` pointer in Figure 4A.1 must be annotated as `CounterAtomic` in a *selective counter-atomicity* design.

`counter_cache_writeback()` function. *Selective counter-atomicity* allows reordering of write accesses (both data and counters) that do not immediately affect consistent recovery of data. However, the programmer needs to ensure that all data and counter values for these writes are persisted to PM before the point in program execution where they start affecting the recoverability. We introduce a function that writes back the programmer-specified counter cache lines, so that the counters for the updated addresses persist to PM *on demand*.

Discussion. The primitives above aims to maximize the performance of the PM systems by trading off programmability, similar to the primitives offered by memory persistency models [1, 3, 40]. The responsibility of their correct usage rests with the programmer. However, we expect that expert-crafted libraries, such as PMDK [35], will abstract away these low-level primitives from regular programmers.

An example of using the primitives. Figure 4A.6 shows an example of using the *selective counter-atomicity* primitives while implementing a transaction with undo-logging. The three stages of the transaction (prepare, mutate and commit in Table 4A.1) are separated by `persist_barrier`

```
 1  struct Backup {
 2      item_t item;
 3      CounterAtomic bool valid;
 4  };
 5
 6  //Undo-logging transaction to modify data
 7  void UndoTx(Backup* log, item_t* data) {
 8      // prepare: creating a valid backup for data in log
 9      PrepareLog(log, data);
10      counter_cache_writeback(log);
11      persist_barrier();
12      // mutate: modify data in-place
13      MutateData(data);
14      counter_cache_writeback(data);
15      persist_barrier();
16      // commit: invalidate backup log
17      log->valid = false;
18      persist_barrier();
19  }
```

Figure 4A.6: Implementation of an undo-logging transaction with selective counter-atomicity primitives.

to make sure the writes from these stages reach PM before the next stage starts. There are two changes in the transaction to leverage the benefits of *selective counter-atomicity.* First, the writes from the prepare and mutate stages do not require strict *counter-atomicity.* Therefore, we allow buffering and reordering of the corresponding data and counter writes. However, before moving on to the next stage of the transaction, we add the `counter_cache_writeback()` function to writeback the latest data and counter values to memory. Second, the write to the `valid` variable in the backup log entry (line 17) invalidates the log entry and commits the transaction. This write access requires *counter-atomicity* as it switches the current consistent data from the log to the modified in-place data. Hence, we annotate the corresponding variable `valid` as `CounterAtomic`.

4A.5 Implementing Selective Counter-Atomicity

In this section, we provide the necessary hardware support to selectively enforce *counter-atomicity.* First, we describe how the *selective counter-atomicity* design is integrated in a system with an encrypted PM. Then, we describe the hardware implementation in the memory controller that enforces counter-atomicity.

4A.5.1 System Integration

Figure 4A.7 shows the high-level overview of a system that supports *counter-atomicity.* On the software side, the programmer annotates the *counter-atomic* variables with `CounterAtomic` primitive and inserts the `counter_cache_writeback()` operations to the program according to the requirement in Section 4A.4. The annotation enables the memory controller to differentiate the *counter-*

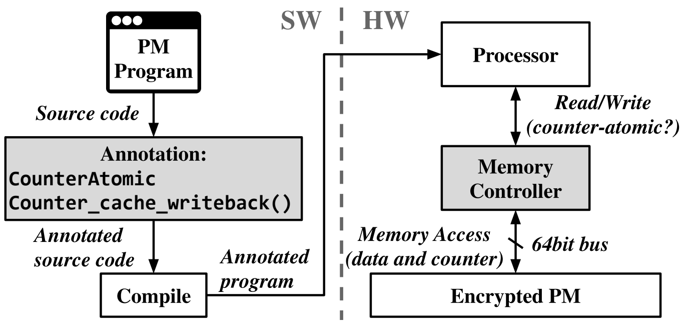


Figure 4A.7: The high-level overview of a system using the *selective counter-atomicity* primitives.

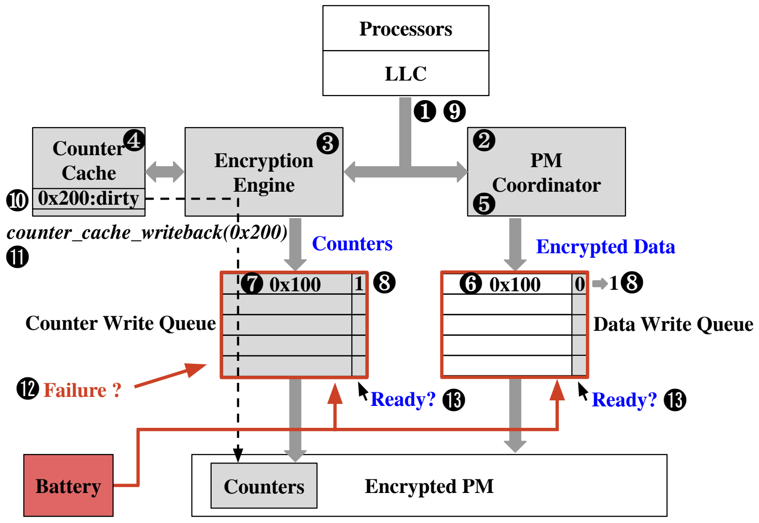


Figure 4A.8: Hardware implementation. The new components are represented with shaded gray, and the persistent structures protected by ADR is shown in red.

atomic writes from the *non-counter-atomic* ones and write back counter cache lines properly. Next, we discuss the hardware support for counter-atomicity.

4A.5.2 Hardware Implementation

Figure 4A.8 depicts the memory controller in our design that supports *(i)* encryption and *(ii)* counter-atomicity. The encryption support consists of an *encryption engine* and a *counter cache*. The counter-atomicity support consists of a *data write queue* and a *counter write queue*. Data encrypted by the *encryption engine* is sent to the *data write queue*, and counters are sent to the *counter write queue*. Next, we describe both encryption and counter-atomicity support in detail.

Encryption and Decryption Support

In this section, we describe the encryption and decryption process in the *encryption engine* using the counters from the *counter cache*, and the necessary steps when the counters are not available in the *counter cache*.

Decryption for Read Accesses. When the processor issues a read request, the *PM coordinator* performs the read access from PM. At the same time, the *encryption engine* accesses the *counter cache* and uses the counter to generate the OTP for the requested memory location, parallelizing the read access and the decryption process. Then, the memory controller decrypts data by XORing the encrypted data and the OTP, completing the read access.

Encryption for Write Accesses. When the processor issues a write request, first, the *encryption engine* generates a new counter by incrementing the global counter, and accesses the *counter cache* to update the stale counter. Second, it generates the OTP with the new counter value. Third, the *PM coordinator* XORs the plaintext data with the OTP and sends the encrypted data to the *data write queue*.

Counter Cache Miss. As our system accesses memory at a cache line granularity, the memory controller fetches a cache line of counters (eight counters) that contains the requested counter from the PM when a read or write access misses the *counter cache*. If a *read* access misses the *counter cache*, it has to stall and wait for the counter to be fetched from the PM. Whereas, if a *write* access misses the *counter cache*, it does not stall, as a new counter that is generated for each write access is used for encryption. After the missing counter cache line is fetched from memory, the encryption engine updates the newly generated counter in the counter cache.

Counter-Atomicity Support

We have shown the hardware support for encrypting and decrypting data. Next, we describe the key mechanisms in the memory controller that guarantee *counter-atomicity*.

Hardware Support for Counter-Atomic Writes. We extend Intel's persistency support to ensure *counter-atomicity* of writes. Intel's persistency support relies on the hardware ADR mechanism that ensures that any write request buffered in the write queue of the memory controller will be written back to PM with some backup power in case of a power failure [43, 204]. Therefore, this mechanism guarantees that any write request that reaches the write queue will always get persisted to the PM. We extend this ADR support to both the *data write queue* and the *counter write queue* and ensure that only the entries that have both the data and associated counter in the write queues get persisted to PM on event of a power failure. To track the data and its counter, we add an extra *ready* bit to each *data write queue* and *counter write queue* entry. The ready bits in both write queues are set *only* when *both* the data and counter writes have been accepted by the corresponding write queues. To make sure any failure does not stop the operation that sets the ready bits in both write queues, this operation is also protected with the ADR support.

A *counter-atomic* write takes three steps to complete. *(i)* The *PM coordinator* sends the encrypted data to the *data write queue*, and at the same time, the *encryption engine* sends the associated counter cache line to the *counter write queue*. *(ii)* When the counter-atomic *data* write reaches the *data write queue*, the memory controller checks whether or not the *counter write queue* has the associated counter entry. If yes, it then sets the *ready* bit in *both* entries to 1. Otherwise, the *ready* bit remains 0. The memory controller performs the same steps when the *counter* from a counter-atomic write reaches the *counter write queue*. *(iii)* Both write queues *only* persist the entries that have the *ready* bit set and any unready entry remains blocked until its ready bit is set. During a system failure, both write queues *only* drain the ready entries. Note that the regular *non-counter-atomic* write queue entries are *always set to be ready.*

The practicality of extending the ADR support. In our evaluated system, we use a 64-entry (4kB) *data write queue* and a 16-entry (1kB) *counter write queue* (hardware overhead details in Section 4A.6.3). The ADR mechanism only has to drain an additional 1kB of *counter write queue* in this case. As future systems are considering flushing the entire processor cache hierarchy (10s of MBs) [204], we believe that our additional overhead is modest and can be implemented in the immediate future. We would like to emphasize that even though our hardware mechanism to enforce *counter-atomicity* relies on the ADR support, in reality, it can be implemented in the hardware using any available hardware mechanism (e.g., hardware logging) that guarantees that the data and counter write queue entries are persistent in case of failure.

Steps During a Counter-Atomic Write. The following is an example of a *counter-atomic* write to the physical address 0x100 (Figure 4A.8). Step ❶: The processor issues a *counter-atomic* write access to the physical address 0x100. Both the *PM coordinator* (step ❷) and the *encryption engine* (step ❸) receive the write. Step ❹: Let's assume that the counter for 0x100 is available in the *counter cache*. The *encryption engine* increments the global counter and updates the counter value in the *counter cache* accordingly. Then it computes the OTP and sends the latest counter to the *counter write queue*. Step ❺: The *PM coordinator* XORs the plaintext data with the OTP generated by the *encryption engine*, and sends the encrypted data to the *data write queue*. Step ❻: The *data write queue* receives the data entry from the *PM coordinator* and checks the *counter write queue* but does not find the counter entry. Therefore, this entry is *unready.* Step ❼: The *counter write queue* receives the counter entry from the *encryption engine* and checks the *data write queue.* Step ❽: As the associated *data write queue* entry has been inserted, the memory controller marks both entries as *ready*, completing the write request.

Steps During a Counter Cache Writeback. Similar to the data cache writeback, the `counter_cache_writeback()` function writes back a user-specified cache line of counters (eight counters) from the *counter cache* to PM without invaliding the cache line, if the requested address hits the *counter cache* and the counter cache line is marked as *dirty.* In this operation, the *ready* bit of the *counter write queue* entry is always set to 1. The following is an example that writes back the counters for the address 0x200 (Figure 4A.8). Step ❾: the processor issues a *counter_cache_writeback()* operation with the address 0x200. Step ❿: The *counter cache* looks up the requested counter cache line and finds that it is *dirty.* As each *counter cache* entry has eight counters, this operation writes back all of them. Step ⓫: The *encryption engine* inserts the *counter cache line* to the *counter write queue.*

Steps During a System Failure. Step ⓬: When a failure occurs, the ADR support gets triggered. Both the *counter and data write queues* start draining the pending write entries. Step ⓭: Both write queues check the *ready* bit and only drain the *ready* entries, making sure that the data and counter in memory are always in sync.

4A.6 Evaluation

In this section, we first describe the evaluation methodology and provide a short description of the evaluated designs, and present detailed evaluation results of each design.

4A.6.1 Methodology

We model the hardware design described in Section 4A.5 in the cycle-accurate simulator Gem5 [205]. The simulated system consists of x86 out-of-order processors, and an 8 GB phase change memory (PCM) [6, 31] with a DDR3 interface (Table 4A.2). Table 4A.2 lists the system parameters used in our evaluation. The *counter cache* in our implementation is 1 MB, 16-way set associative. As each counter is 8 B, a 1 MB counter cache can store 128 k counters. However, we show results with different sizes of counter cache in Section 4A.6.3.

Table 4A.2: System configuration. Tests are single-thread and single-core unless explicitly mentioned.

Processor	Out-of-Order Cores, 4.0 GHz
L1 D/I cache	64 kB/32 kB per core (private), 8-way
L2 cache	2 MB per core (shared), 8-way
Counter cache	1 MB per core (shared), 16-way
Memory controller	Data read/write queue: 32/64 entries Counter write queue: 16 entries
Memory	8 GB PCM, 533 MHz [40], $t_{RCD}/t_{CL}/t_{CWD}/t_{FAW}/t_{WTR}/t_{WR}$ = 48/15/13/50/7.5/300 ns [6]
En/decryption	40 ns latency [48]

The following are the evaluated designs:

- **No-encryption design.** A PM system without any encryption.

- **Ideal design.** An encrypted and crash consistent PM system using the counter-mode encryption technique but without any *counter-atomicity* overhead.

- **Co-located data and counter design (Co-located).** An encrypted and crash consistent PM system using a 72-bit memory bus, where the counter used for encryption is co-located with the corresponding data within each cache line (Section 4A.3.2).

- **Co-located data and counter with a separate counter cache design (Co-located w/ C-Cache).** Similar to the prior design with a wider memory bus (Co-located), but the counters are separately buffered in the *counter cache* and written back to PM using one access co-locating both the data and counter (Section 4A.3.2). Note that these two designs require adding extra pins in the memory bus, which is expensive.

- **Full counter-atomicity design (FCA).** An encrypted and crash consistent PM system with the existing memory bus, where *counter-atomicity* is enforced for every write operation using our proposed hardware mechanism in the memory controller (Section 4A.3.2).

- **Selective counter-atomicity design (SCA).** Similar to the previous design (FCA). However, writes are *counter-atomic* only when necessary (Section 4A.4).

Next, we describe the implement details of Intel's persistency support in our simulation environment. The implementation requires two supports. *(i)* Hardware support for the `clwb` instruction that writes back cache lines. *(ii)* Hardware support for `sfence` that ensures that any store instruction preceding the `sfence` instruction in the program order completes before any store instruction that comes after the fence. First, we model the `clwb` instruction in the simulator by writing back the user-specified cache lines to PM without invalidating them. Second, we implement the support for `sfence` by ensuring that all outstanding `clwb` instructions are completed before an `sfence` instruction can retire. We instrument our workloads with `clwb` and `sfence` instructions in the appropriate places.

4A.6.2 Workloads

We evaluate five PM workloads that manipulate different persistent data structures. Our evaluated workloads are similar to the ones used in prior works on persistent memory systems [37, 40, 47, 197].

- **Array Swap.** Swaps random items in a persistent array.
- **Queue.** Randomly en/dequeues items to/from a persistent queue.
- **Hash Table.** Inserts random values to a persistent hash table.
- **B-Tree.** Inserts random values into a persistent B-tree.
- **Red-Black Tree.** Inserts random values into a persistent red-black tree.

4A.6.3 Results

We first evaluate the impact of different designs (listed in Section 4A.6.1) on performance and throughput. Then we compare the write traffic in these designs. Last, we evaluate the sensitivity of the results when we vary different parameters.

Single-Core Performance.

In this experiment, we compare the performance improvement of different designs. Figure 4A.9 demonstrates the runtime of different design point normalized to the *no-encryption* design. The observations are as follows. First, the *selective counter-atomicity* design improves performance on average by 6.3% over the *full counter-atomicity* design, and is only 11.7% slower than the *no-encryption* design (due to the benefit from reordering and buffering of writes). Second, the co-located design without any counter cache significantly slows down the performance, on average 81.1% slower than the *selective counter-atomicity* design (due to the serialized read and decryption). The co-located design with a *counter cache* is slightly faster than the *selective counter-atomicity*

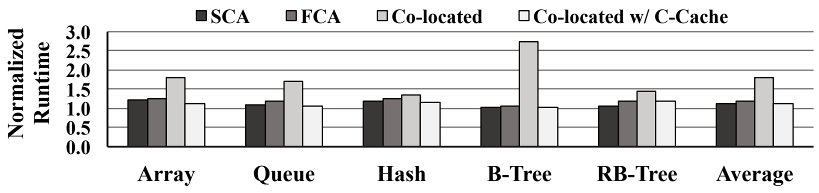


Figure 4A.9: Performance comparison of different design points. The runtime is normalized to the *no-encryption* design (lower is better).

design (0.7% faster), and only degrades the performance on average by 10.9% compared to the *no-encryption* design. However, co-locating the data and counter is impractical due to invasive changes in the memory subsystem. We conclude that using *selective counter-atomicity* is an efficient and practical design that guarantees crash consistency of an encrypted PM system.

Multi-Core Performance

This experiment evaluates different design points in a multi-core system, where each thread performs the same operations on different cores. Figure 4A.10 demonstrates the throughput of different designs. For each workload, the x-axis shows the number of cores, and the y-axis shows the throughput (number of transactions per second) normalized to the single-core *no-encryption* design. We make the following observations. First, the throughput of the *selective counter-atomicity* design is very close to that of the *ideal design* and is significantly better than the *full counter-atomicity* and the *co-located* design. As the number of cores increases, the benefit of selective counter-atomicity over full counter-atomicity also increases. In a 1/2/4/8-core system, *selective counter-atomicity* improves performance on average by 6.3/11.5/21.8/40.3% over *full counter-atomicity.* On the other hand, the throughput of *selective counter-atomicity* comes within 4.7% of the *ideal design* in all system configurations. Therefore, we conclude that *selective counter-atomicity* is highly scalable compared to other designs. Second, the co-located design with a counter cache has similar performance as the *selective counter-atomicity* design, as they use the same counter cache. However, it performs better in some workloads in four and eight core configurations because the co-located designs use a wider memory bus (72 bits instead of 64 bits) and faces less congestion on the memory bus. Third, we notice that two workloads (Queue and RB-Tree) exhibit relatively poor scalability with *selective counter-atomicity.* We find that there is a high fraction of counter-atomic writes in their data structures, leading to contention in the memory controller. We conclude that selective counter-atomicity ensures scalability without changing the memory interface.

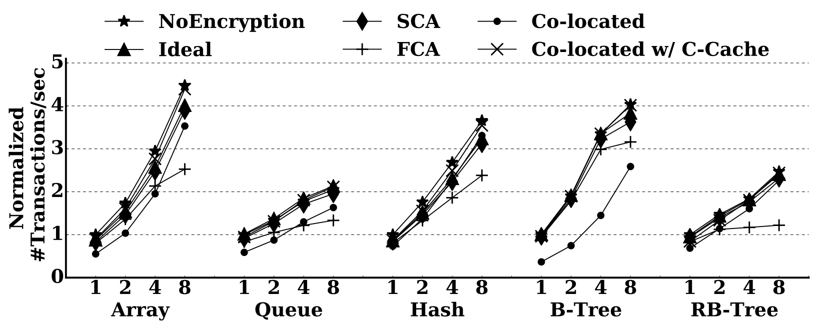


Figure 4A.10: Throughput of multithreaded workloads, normalized to the single-core *no-encryption* design (higher is better).

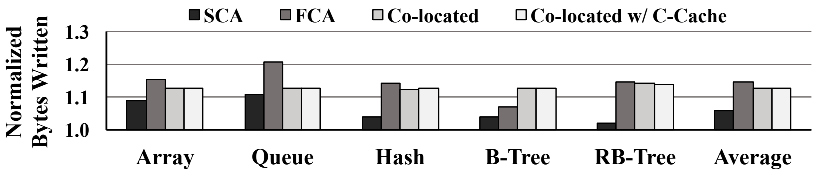


Figure 4A.11: Write Traffic to PM normalized to the *no-encryption* design (lower is better).

Write Traffic

Figure 4A.11 shows the write traffic to the PM normalized to the *no-encryption* design. We first observe that *selective counter-atomicity* on average reduces the write traffic by 8.1% compared to the *full counter-atomicity* design. This is because *selective counter-atomicity* buffers and coalesces the counter updates and writebacks at the end of a transaction, therefore, reduces the counter write traffic to PM. Second, the write traffic in designs that co-locate data and counters are similar as they enforce counter write back together with every data write to memory. The *selective counter-atomicity* design reduces the write traffic on average by 6.6% compared to these two designs.

Reducing the write traffic not only provides better performance, but also improves the lifetime of PM. Selective counter-atomicity can improve the PM lifetime by 6.6% assuming a uniform wear-leveling technique [54] (an orthogonal design consideration in the PM systems). The improvement will be higher if we consider compressing the counters using techniques proposed by some prior works [80, 83].

Sensitivity to Counter Cache Size

Figure 4A.12 compares the performance of the *selective counter-atomicity* design when we vary the counter cache size from 128 kB to 8 MB and run workloads with footprints ranging from 100 MB to 1000 MB. Figure 4A.12a and 4A.12b show the average speedup and miss rate with different counter cache sizes over the smallest 128 kB counter cache. We observe that as the size of the counter

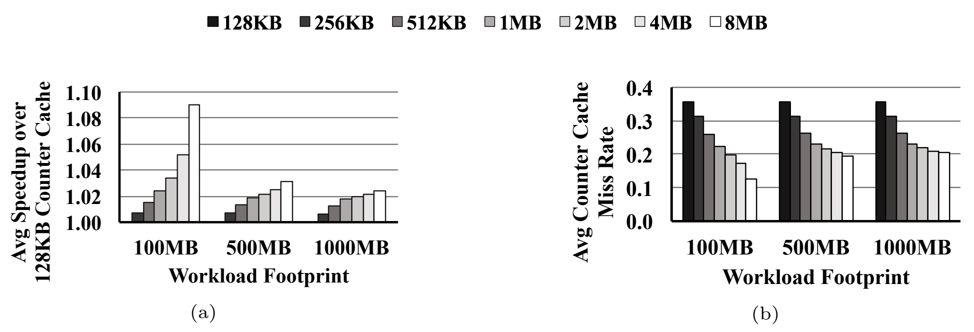


Figure 4A.12: Evaluating SCA with different sizes of counter cache. (a) Average speedup over a 128 kB counter cache (higher is better). (b) Average counter cache miss rate (lower is better).

cache increases, both the speedup and miss rate improve for all workloads. While increasing the footprint of the workload decreases the benefit from a larger counter cache. For example, an 8 MB counter cache improves the performance by 9% over a 128 kB counter cache when the workloads have 100 MB footprint. On the other hand, the improvement is only 2.4% with 1000 MB workloads. Similarly, using an 8 MB counter cache decreases the miss rate by 23.3% with a 100 MB workload, while the miss rate decreases by 15.4% with a 1000 MB workload. We conclude that using a large counter cache lead to better performance, but as the footprint of workload increases, the performance becomes less sensitive to the counter cache size. In this work, we evaluate a 1MB counter cache per core, similar to a prior PM encryption work [199].

Sensitivity to Transaction Size

In this experiment, we evaluate the overhead of *selective counter-atomicity* with variable transaction size. Figure 4A.13 compares the performance of *selective counter-atomicity* when varying the transaction size from 64 B to 4 kB. The x-axis shows the number of cache lines committed at each transaction. The y-axis shows the runtime normalized to the corresponding *ideal design* that do not enforce counter-atomicity. We observe that when the transaction size is small, the overhead of *selective counter-atomicity* is on average 7.5%. The overhead decreases as the size of transaction increases, and becomes less than 1% in all cases when processing transactions with a size similar to a page (4 kB). Specifically, the overhead becomes as low as 0.1% for the B-Tree. As the size of the transaction increases, the fraction of *counter-atomic* write gets smaller, which amortizes the overhead of *counter-atomicity.* We conclude that the overhead of *selective counter-atomicity* will be negligible in PM applications that manipulate a large dataset within a transaction.

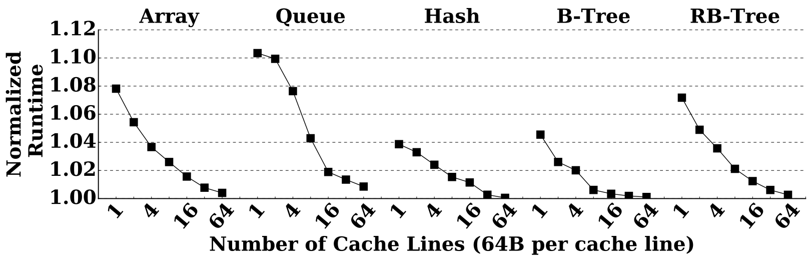
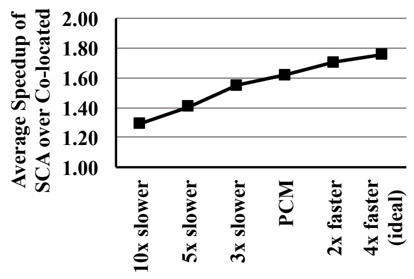


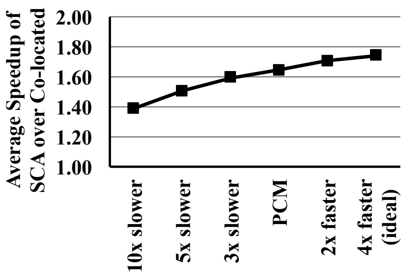
Figure 4A.13: The runtime of SCA with different sizes of transaction, normalized to the ideal design (lower is better).

Sensitivity to PM latency

In this experiment, we compare the performance of *selective counter-atomicity* with the design that co-locates both the counter and data (Section 4A.3.2) with varying PM latency (Figure 4A.14) to understand the performance sensitivity of *selective counter-atomicity* to different PM technologies. First, we keep the write latency fixed (same as the PCM latency) and vary the read latency from 10× slower to 4× faster (similar to the DRAM latency). Then we keep the read latency fixed and vary the write latency in a similar way. We have the following conclusions from the results. First, Figure 4A.14a shows that as read latency decreases, *selective counter-atomicity* is on average 29.3% to 75.6% faster than the co-located design. This is due to the fact that the serialized decryption overhead in the co-located design becomes more prominent with a lower read latency and therefore, by parallelizing the memory read access and the decryption process, *selective counter-atomicity* provides better performance. Second, Figure 4A.14b shows that *selective counter-atomicity* is on average 38.9% to 74% faster than the co-located design when we decrease the write latency. The reason is that the performance of the co-located design is not very sensitive to the write latency, as writes are usually not on the critical path and it uses a wider bus to writeback data and counter atomically. However, *selective counter-atomicity* needs to writeback the counters through the same bus as data, and therefore, lowering the write latency provides a significant benefit by reducing the bandwidth contention between data and counters. Third, *selective counter-atomicity* provides a significant performance benefit (29.3%/38.9% for read/write) even when the PM speed is 10× slower than the PCM. It demonstrates that *selective counter-atomicity* is effective, even when the write latency is very high. We believe that future systems will optimize PM for lower latency and higher throughput, hereby adopting *selective counter-atomicity* will lead to better performance.



(a) Read Latency



(b) Write Latency

Figure 4A.14: Varying (a) read latency, and (b) write latency.

Overhead Analysis

Finally, we analyze the overhead from the additional structures used to provide *selective counter-atomicity*. Similar to the prior hardware memory encryption techniques, we use a counter cache and an encryption engine [48, 52, 199]. The size of our counter cache is 1 MB per core, similar to the one employed by Awad et al. [199]. Our proposed design, *selective counter-atomicity*, requires only an additional 16-entry (1 kB in size) *counter write queue* at the memory controller.

4A.7 Discussion

This paper targets a non-volatile memory that has similar read latency and slower write latency compared to DRAM, as adopted in prior work [1, 6, 31, 36, 37, 39, 40, 45, 46, 125, 126, 135, 158, 197, 199, 206–210]. The true potential of a persistent memory can be exploited when the PM is on the memory bus with a low access latency. In these cases, the encryption latency becomes a significant bottleneck in memory read accesses and therefore, it is essential to optimize this overhead by parallelizing read and decryption with cached counters [48, 50]. However, the commodity PM chips are yet to become commercialized in a wide scale and the current PM products still place PM over the PCIe bus with a latency close to high-end SSDs [211–215]. We do not evaluate these systems as the encryption latency is small compared to the overall access latency. We believe that the future systems will perfect the PM technologies over time and harness its true potential by placing PM on the memory bus. Our proposed technique to optimize *counter-atomicity* would be highly valuable in those systems.

Chapter 4B

Optimizing Memory and Storage Support for Persistent Memory Systems

4B.1 Introduction

As Chapter 3 has described, crash-consistent software for PM systems exhibit a unique property—they place *writes to memory on the critical path* of program execution. For conventional software, only reads to memory are on the critical path, while writes may be buffered, coalesced and reordered on the way to memory for better performance. However, for crash-consistent software, the order of writes to memory is severely constrained to ensure data recoverability across failures [1,25,39,40,135]. Furthermore, crash-consistent software often has to guarantee the durability of data. For example, programmers executing a database transaction expect that data modified within a transaction becomes persistent when the transaction completes. Therefore, all the writes issued to persistent data within a transaction have to reach all the way to PM (or more specifically, the persistent domain) before the transaction completes. The x86 and ARM ISA introduced new instructions [3, 34] that programmers can use to ensure that writes reach the persistent domain to provide durability guarantees required in crash-consistent software. However, these durability guarantees imply that writes to persistent data fall on the critical path of program execution.

Placing recoverable data on PM not only moves writes onto the critical path, it further degrades performance by increasing the latency of writes. The latency increases due to additional constraints on maintaining persistent data in PM. For example, the PM encryption operation introduced in Chapter 4A places the encryption latency onto the write path. Likewise, protecting the integrity of persistent data using integrity verification techniques further adds on additional operations that for verification [50, 54, 55, 57–59]. Furthermore, most PM technologies suffer from a limited bandwidth and wear out after a certain number of writes, necessitating deduplication, compression, and/or wear-leveling of PM writes [73–77, 87]. All these encryption, integrity protection, deduplication, and compression operations, collectively referred to as *backend memory operations (BMOs)* henceforth, are performed at the memory controller and significantly increase the PM write latency. Moreover, since writes fall on the critical path of the crash-consistent software, the increase in write latency significantly degrades application performance. In this work, our *goal* is to minimize the latency overhead in write operations caused by these BMOs in PM systems.

The key challenge here is in figuring out *how to optimize these seemingly dependent, monolithic operations*. When viewed as dependent, indivisible operations, common latency optimizations (e.g., parallelization) are precluded. For example, in a system with encryption and compression, performing compression before encryption is a reasonable approach, while performing them in parallel is not, as compression can change the address mapping of the compressed data which will then invalidate the encryption output that used the old address. Our key insight to optimize these BMOs is that when they are viewed as monolithic, indivisible operations, they have to be performed in series, however, *if each BMO is viewed as a series of sub-operations, there exist many opportunities to optimize individual sub-operations across BMOs*.

By viewing each BMO as a series of sub-operations, we can optimize them for latency using two mechanisms: (i) *parallelization* of sub-operations across BMOs and (ii) *pre-execution* of sub-operations without waiting until the PM write reaches the memory controller. First, when BMOs are viewed as a series of sub-operations, there are many opportunities for parallelization as some sub-operations across BMOs do not have any dependencies among them and can be executed in parallel. For example, even though deduplication should happen before encrypting data, the first sub-operation of deduplication calculates and looks up the hash of the data value in the write request and can be executed in parallel with the first sub-operation of PM encryption that uses the address of the write to generate a one-time pad (details in Section 4B.3.1).

Second, while parallelization of the sub-operations helps speeding up the BMOs, we observed that significant performance gains are still left on the table. Our key observation is that the parallelized approach does not start any of the sub-operations until the write access reaches the memory controller, however, the inputs necessary for the sub-operations are available much earlier in practice. For example, undo-logging [35, 37, 125, 130] is frequently used in crash consistent PM programs. An undo-logging transaction creates a backup copy of the data before modifying it. Before the modification takes place, the address and data for modification are already known during the backup step. Therefore, the BMOs for the update can be pre-executed as soon as the data and address become known at the backup step. We categorize sub-operations as address-dependent, data-dependent, or both. They can be *pre-executed* as soon as the address and/or data is available. Pre-execution of these sub-operations *decouples* them from the original write and moves them off the critical path, delivering a significant performance gain.

Based on these two key ideas, we introduce Janus, a generic and extensible framework that parallelizes and pre-executes BMOs in PM systems by decomposing them into smaller sub-operations. It provides a hardware implementation for parallelization and pre-execution, and exposes an interface to the software to communicate the address and data values of write requests before the write reaches the memory controller. However, several challenges need to be addressed both in the design of the hardware and the software interface of Janus. The challenges in the hardware design are as follows: First, the pre-executed results of the various sub-operations for the individual writes should not change the processor or memory state until the corresponding write operation happens. Second, the pre-execution should not be dependent on any *stale* processor or memory state to maintain correctness of the results. To address these challenges, we maintain an *intermediate result buffer (IRB)* in the memory controller to store the pre-executed results and isolate them from any other processor or memory state. We also track the address and data of the write operations in IRB to detect and invalidate any stale pre-execution results.

The challenges in the software interface design are the following: First, with PM still being in a nascent stage of adoption, the software interface must be generic and extensible to systems with different BMOs. We only expose the address and the input data in the interface, decoupling the interface from the BMOs implemented in the system. Second, the software interface needs to be easy-to-use and applicable to different PM-based programs. We address this issue by providing a variety of functions that are suitable for different PM programming models. We show that the frequently used crash-consistent software mechanisms, such as undo-logging, are particularly amenable to leveraging

Janus's pre-execution interface and programmers can manually insert these pre-execution requests to gain significant performance improvement. However, we also provide a compiler pass to automatically instrument the source code to alleviate programmer's burden. We describe our design and our proposed solutions to these challenges in Section 4B.4.

The contributions of this work are as follows:

- We show that it is possible to optimize the BMOs in PM systems by decomposing these seemingly monolithic, dependent operations into a series of *sub-operations*.
- We propose a generic and extensible solution to optimize the sub-operations by categorizing their dependencies. First, we show that independent sub-operations across BMOs can be executed *in parallel*. Second, we show that the sub-operations can be *pre-executed* as soon as their inputs are available, which moves the latency of BMOs off the critical path of the writes.
- We propose Janus, the first system that parallelizes and pre-executes BMOs before the actual write takes place. Janus provides a generic interface that decouples different BMOs at the hardware from the software.
- We exhibit the effectiveness of Janus by evaluating a PM system that integrates encryption, integrity verification, and deduplication as the BMOs in the hardware. Our experimental results show that Janus achieves on average a 2.35× speedup while executing a set of applications where pre-execution requests are inserted manually over a baseline system that performs the BMOs serially. In comparison, instrumenting programs by our automated compiler pass achieves on average a 2.00× speedup over the serialized baseline.

4B.2 Performance Overhead of BMOs

Section 2.3.3 has introduced various memory and storage support operations, i.e., BMOs, that make PM more secure and robust. A major challenge is that these operations add extra latency to writes, as they all require certain computation or cache lookup before actually performing the write access. To maintain the correctness, BMOs should follow certain dependencies among themselves. For example, a system with encryption, deduplication and integrity verification, these BMOs should happen in the order of deduplication, encryption, and integrity information update during a write access. Deduplication first tells whether the write is necessary or not. Then, the encryption engine

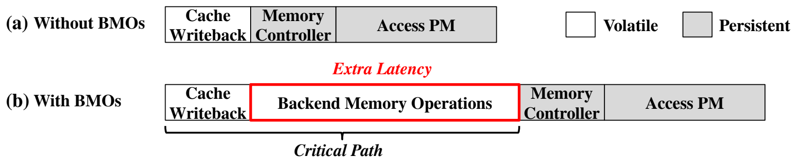


Figure 4B.1: Write latency (a) without and (b) with BMOs.

encrypts the data if the write is not a duplicate. Finally, the integrity mechanism (e.g., Bonsai Merkle Tree [29]) updates the message authentication code (MAC) and hash tree to protect the encrypted data and counters. The ordering constraints serialize the latency from different BMOs, which is in the order of hundred nanoseconds.

Figure 4B.1 demonstrates the latency breakdown of a write access. We assume a system with the Intel Asynchronous DRAM Refresh (ADR) technique [43] that ensures the write queues are in the persistence domain. Therefore, writes to PM become persistent (or non-volatile) as soon as they are placed in the write queue in the memory controller, as the ADR technique can flush the write queue to PM in case of a crash. Without any BMOs (Figure 4B.1a), only the writeback from the cache hierarchy to the memory controller is exposed on the critical path, which typically takes around 15 ns in modern processors (e.g., Intel i7 processor [216]). The subsequent operations in the memory controller and the actual PM device write operations do not contribute to the critical write latency. However, with BMOs (Figure 4B.1b), both the writeback and the BMO latency are exposed on the critical path, as until BMOs are completed, the write cannot be placed in the write queue and hence cannot be considered persistent. As these BMOs add extra hundreds of nanoseconds of latency, the critical latency increases by more than 10 times.

4B.3 Overview

In this section, we describe our key ideas in optimizing the BMOs and providing the crash consistency guarantee.

4B.3.1 Key Ideas

The BMOs need to execute in series if we regard them as monolithic, indivisible operations. However, we observe that BMOs can be further decomposed into smaller *sub-operations*. We first demonstrate decomposing two commonly used BMOs in PM systems: counter-mode encryption [28, 48–50, 53, 56–58] and deduplication [73–77]. Next, we take a two-pronged approach to minimizing their latency:

(1) parallelize BMOs as much as possible, and (2) pre-execute BMOs to move their latency off the critical path.

Decomposing BMOs. The counter-mode encryption [17] is an efficient encryption scheme that indirectly encrypts data blocks using unique counters. Its hardware implementation typically encrypts a unique counter together with the address of the data block into a bitstream called one-time padding (OTP), and then it XORs this bitstream with the data block to complete the encryption. To accelerate the read access, the hardware mechanism buffers these counters in an on-chip counter cache so that decryption can begin without waiting for data to be fetched from PM, reducing the read latency. During a write access, it performs three sub-operations : (E1) generate a new counter, (E2) generate the one-time padding: `OTP = En(counter|address)`, and (E3) encrypt data with an XOR operation: `EncData = OTP` $\oplus$ `Data`. As encryption begins only when both the data and address of the write access reaches the encryption engine, the whole latency is added to the write access.

On the other hand, the deduplication mechanism detects whether writes contain a value that already exists in memory and cancels the write if a duplicated value is found. The hardware mechanism maintains a deduplication table that stores the hashes (fingerprints) of existing data blocks to detect duplicates, and an address mapping table to redirect the writes to the existing copy of data in memory. During a write access, a deduplication operation consists of four sub-operations: (D1) hash data, (D2) lookup the hash value in the deduplication table, (D3) update the address mapping table, and (D4) encrypt the new address mapping table entries and writeback to PM. To integrate encryption and deduplication, We assume a scheme similar to DeWrite, where the counter and deduplication address mapping co-locate in the same metadata entry [77]. Next, we describe the parallelization and pre-execution of the decomposed BMOs.

Parallelization. We observe that there are two types of dependencies between the previously decomposed sub-operations: *intra-operation dependency* and *inter-operation dependency.* Intra-operation dependency describes the dependency between sub-operations within one BMO, while, inter-operation dependency describes the dependency between sub-operations between different BMOs. We demonstrate the dependencies as a dependency graph in Figure 4B.2a. Two sets of sub-operations can happen in parallel as long as there is no incoming inter- or intra-operation dependency path from one set to another. Formally, let a node of sub-operation be Op, a set of Op be S, and a path from Op_1 to Op_2 be $Op_1 \rightsquigarrow Op_2$, S_1 and S_2 can execute in parallel, i.e., $S_1 \parallel S_2$,

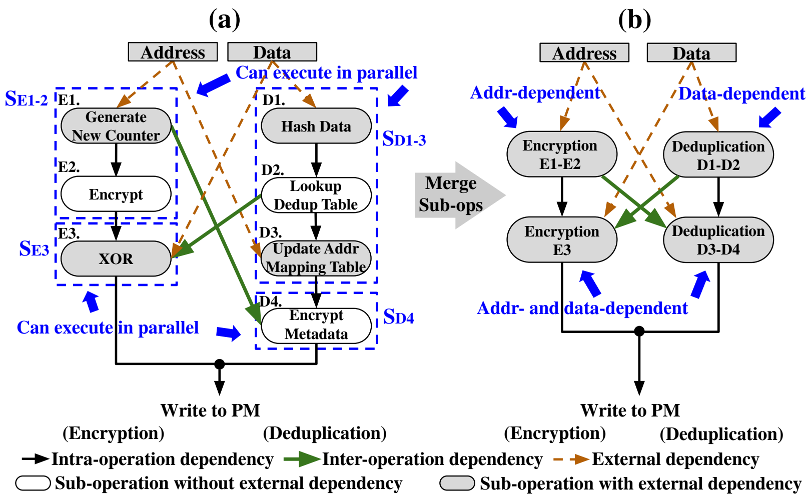


Figure 4B.2: Optimize encryption and deduplication by: (a) parallelizing sub-operations, and (b) categorizing sub-operations by external dependency for pre-execution.

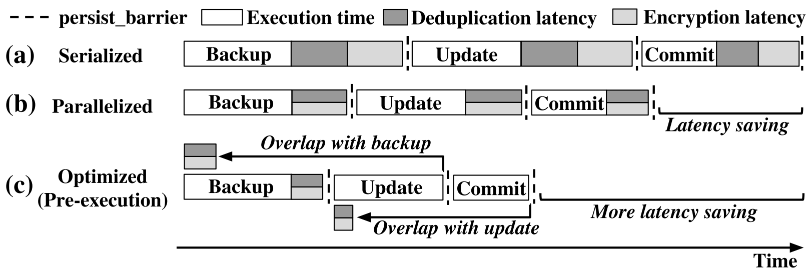


Figure 4B.3: Timeline of an undo log with (a) serialized, (b) parallelized and (c) pre-executed BMOs.

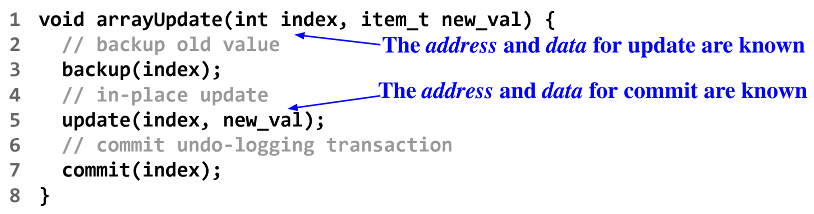
if and only if $\forall Op_1 \in S_1$ and $\forall Op_2 \in S_2, \nexists Op_1 \rightsquigarrow Op_2 \wedge \nexists Op_2 \rightsquigarrow Op_1$.

Next, we apply our theory to the example in Figure 4B.2. We mark the intra- and inter-operation dependencies with black and green edges, respectively. The intra-operation dependencies in each BMOs follows the order of steps. And, there are two inter-operation dependencies: D4 depends on E1 as the address mapping co-locates with counter, and E3 depends on D2 as the memory controller cancels duplicated writes. According to these dependencies, we can circle out the sub-operations that are independent in each backend operations (blue boxes): S_{E1-2} and S_{D1-3} are independent, and S_{E3} and S_{D4} are independent. Therefore, they can be executed in parallel. By parallelizing groups of sub-operations, we reduce the serialization overhead. Figure 4B.3a shows the execution timeline of an undo-logging transaction that consists of three steps (each with PM writes): backup, update, and commit. In the serialized approach, deduplication and encryption operations are serialized for each step of the transaction. However, by parallelizing independent sub-operations, the execution latency of the three steps in an undo-logging transaction can be reduced, as shown in Figure 4B.3b.

Pre-execution. So far, we have exploited the parallelism between BMOs by decomposing them into sub-operations. We further observe that the BMOs process two types of external inputs: the data and address of a write. These external inputs are different from any intermediate inputs generated and used between different sub-operations of the same BMO. Accordingly, apart from the inter- and intra-operation dependencies introduced earlier, external inputs introduce a new dependency, *external dependency* (marked as yellow arrow), that takes into account the external input of each sub-operation. A sub-operation is dependent on an external input if there exists an external dependency edge from the input. We merge nodes without any external dependency (marked in white) with their preceding nodes with external dependency (marked in gray). Figure 4B.2b shows the simplified graph after the merge operation. A set of merged sub-operations is externally dependent on an external input if there exists an external dependency edge from the input or a path that indirectly connects the input to one/some of its sub-operation node (via inter- and intra-operation dependency edges). Formally, let the set of merged sub-operation nodes be S, an input (address or data of a write) be In, then S has an external dependency to In if and only if $\exists Op \in S, In \rightsquigarrow Op$.

Based on the type of external input, we categorize sub-operations into three types: address-dependent, data-dependent, and address- and data-dependent. In the example of Figure 4B.2b, E1-E2 are address-dependent, D1-D2 are data-dependent, and E3 and D3-D4 are both address- and data-dependent. The external dependency implies that as soon as the external inputs are available, the BMOs can start execution even before the actual write access reaches the memory controller. Next, we use a code example to explain how we can exploit the opportunity of pre-executing BMOs.

Example. Figure 4B.4 shows an example of updating an array using an undo-logging transaction that follows three steps: backup the old data, perform the in-place update, and commit the update. In this example, the address and data for the in-place update are known before the backup step (at line 1). Similarly, the address and data for the commit (validate the in-place update) are known before the commit step (at line 5). Therefore, the pre-execution of the BMOs for the in-place update and the commit steps can be overlapped with the previous steps of the undo-logging transactions, moving them off the critical path. Figure 4B.3c shows the timeline of this pre-execution. By pre-executing the BMOs that have already been parallelized, we can gain a significant speedup.



```
void arrayUpdate(int index, item_t new_val) {
  // backup old value
  backup(index);
  // in-place update
  update(index, new_val);
  // commit undo-logging transaction
  commit(index);
}
```

The *address* and *data* for update are known
The *address* and *data* for commit are known

Figure 4B.4: An example of pre-executing BMOs in an undo-logging transaction.

4B.3.2 Requirements

Pre-executing the BMOs before the actual write happens provides a significant benefit. However, the pre-execution should not affect the correctness of the normal execution. We summarize the requirements on the *hardware* support for pre-execution as follows:

1. Does not affect processor state. The pre-execution should not affect the processor or memory state, i.e., it should not change the data or metadata in memory, cache or register files.

2. Invalidates stale pre-execution results. The pre-execution should not be dependent on a stale processor or memory state. i.e., if the processor or memory state used in the pre-execution has been modified before the actual write access, the pre-execution result becomes invalid.

On the other hand, we need to provide an interface to let the software leverage the hardware support. We summarize the requirements on the *software* interface as follows:

3. Extensible interface. The software interface needs to be generic and extensible to systems with different BMOs, i.e., programs developed with the same interface should be compatible even though the BMOs change in the hardware.

4. Programmable. The software interface needs to be easy-to-use for different programming models that ensure crash consistency (e.g., undo /redo/shadow logging), and should abstract away the memory layout.

Section 4B.4.3 presents our solution to meet the two requirements for the hardware support, and Section 4B.4.4 presents our software support that meets the two requirements on the software interface.

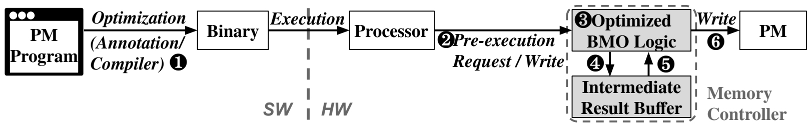


Figure 4B.5: High-level of Janus (HW changes are shaded).

4B.4 JANUS

In this section, we first describe the high-level design of our proposed system and then provide the details of the hardware mechanism (Section 4B.4.3) and software support (Section 4B.4.4).

4B.4.1 High-level of Janus

The *goal* of this work is to reduce the overhead of BMOs in write accesses using a software-hardware co-design. Figure 4B.5 shows an overview of Janus. On the software side, programmers annotate the PM programs using our software interface (step ❶). To further reduce the programming effort, we provide a compiler pass that automatically instruments the program. We present the use of Janus interface in Section 4B.4.4 and the design of our compiler pass in Section 4B.4.5. On the hardware side, the processor issues pre-execution requests to the memory controller during the execution of the annotated programs (step ❷). The processing of pre-execution requests consists of two parts. First, the *optimized BMOs logic* of Janus executes the sub-operations of the requests in parallel (step ❸). Then, it stores the temporal results in the *intermediate result buffer* (step ❹). When the actual writes arrive at the memory controller, they do not need to go through the BMOs, instead, they use the pre-executed results from the *intermediate result buffer* (step ❺) and complete the access to PM (step ❻).

In the rest of this section, we first introduce the integration of three common BMOs in PM systems. Then, we present Janus hardware details in Section 4B.4.3, and the software interface in Section 4B.4.4. Finally, we discuss the solutions to potential exceptions when integrating Janus in real systems in Section 4B.4.6.

4B.4.2 Backend Memory Operations

BMOs are integrated into memory and storage systems for different purposes, such as ensuring confidentiality and integrity, improving the lifespan, mitigating the write bandwidth limitation, etc. If we treat each of them as an entity, it seems difficult to execute them in parallel as the output of one operation flows into another. However, by breaking them down into smaller steps, we can

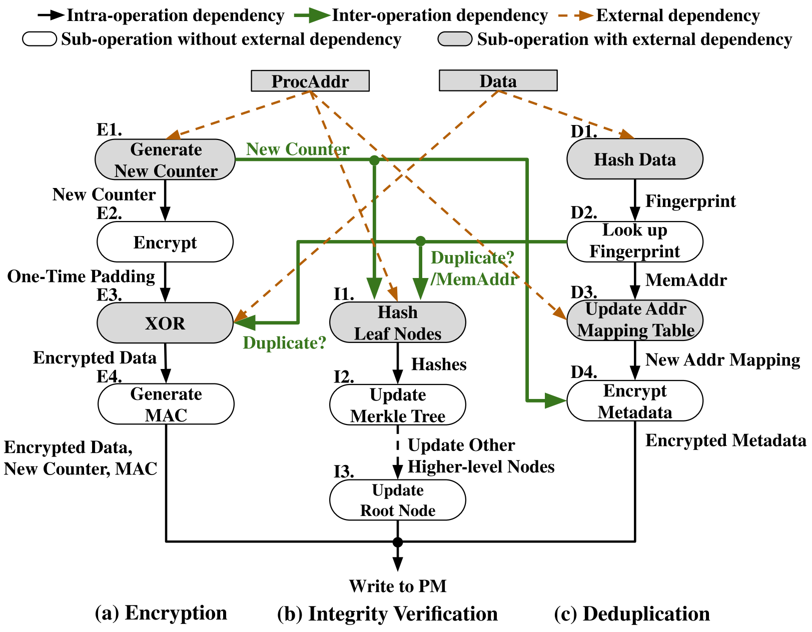


Figure 4B.6: The dependency graph of backend operations.

leverage the underlying parallelism. There has been a myriad of BMOs, as shown in Table 2.1. To better demonstrate our idea, we take the two BMOs introduced in Section 4B.3: encryption and deduplication, together with another popular BMO, integrity verification. Figure 4B.6 presents the break down of the three BMOs.

As we already described the operations in encryption and deduplication in Section 4B.3, here we introduce the steps in an integrity verification technique. The Bonsai Merkle Tree [29] is an integrity verification scheme designed for memory encrypted under counter-mode. The leaf nodes of the tree are counters and the intermediate nodes are hashes of their child nodes. Therefore, the root hash is essentially the hash of all leaf nodes. Keeping the root hash in a secured non-volatile register ensures the integrity of the entire memory [29, 57]. Each data block is protected by a message authentication code (MAC) that consists of the encrypted data and its counter, i.e., `MAC = Hash(EncData, Counter)`. During a read accesses, the integrity verification mechanism compares the root hash computed from the counter read from memory with the existing root to verify the integrity. During a write access, this mechanism updates the integrity information in the following steps (Figure 4B.6b): First, the integrity verification mechanism computes the hash of leaf nodes (step I1), and then it keeps computing higher-level intermediate nodes *all the way to the root* (step I2-I3). The intra-operation dependencies between these steps are indicated by black arrows. In this mechanism, the write access includes this long latency of hashing. For example, if we assume each intermediate node is the hash of eight lower-level nodes, then the height of the Merkle Tree is 9 in a system with only 4GB PM, resulting in a 360 ns latency for each write.

The integration of integrity verification with the other two BMOs introduces an extra step: the encryption operation needs to compute the MAC for Integrity verification before writing data back to PM (step E4). Similar to the prior work, DeWrite [77], the Merkle Tree in our mechanism is built on the co-located address mapping and counter so that the metadata storage can be minimized. Therefore, the integration also introduces new inter-operation dependencies (green edges). The integrity verification support needs to take the latest counters or the remapping address (if duplicate) to update the Merkle Tree. Thus, step I1 depends on E1 and D2 (edge E1→I1 and D2→I1). To mitigate the extra latency on writes, we first apply the rule for *parallelization* (Section 4B.3.1). Based on the intra-operation dependency edges, three sets of sub-operations E3-E4, I1-I3 and D3-D4 can execute in parallel as there is no edge between any pair of these sub-operation sets. Then, we apply the rule for *pre-execution*. We mark the nodes with external-dependency in gray. After merging the nodes without external-dependency (marked in white) to the ones with external-dependency, we find out that E1-E2 are address-dependent, D1-D2 are data-dependent, and the rest are both address- and data-dependent. These regions can be pre-executed once the dependencies are resolved. Next, we describe the hardware support that enables pre-execution.

4B.4.3 Hardware Support

In this section, we describe the hardware support that meets the two requirements that we outlined in Section 4B.3.2.

Hardware Support for Pre-execution

Does not affect processor state. The pre-execution of BMOs should not change the processor or memory state. Therefore, Janus stores the temporary results in an *Intermediate Result Buffer (IRB)*. Figure 4B.7c shows the fields in each IRB entry (and their sizes). IRB needs to support two basic functionalities: identify different pre-execution requests, and store and provide the pre-executed results. First, Janus uses a `PRE_ID` for each request in order to make sure the pre-execution requests are unique. Considering that different threads can be executing the same program and can have the same `PRE_ID`, each entry contains another field, `ThreadID` that distinguishes the requests from different threads. As recent works have proposed deferred commit [125, 166], a transaction may not have all the updates written back to PM before the transaction completes, causing more

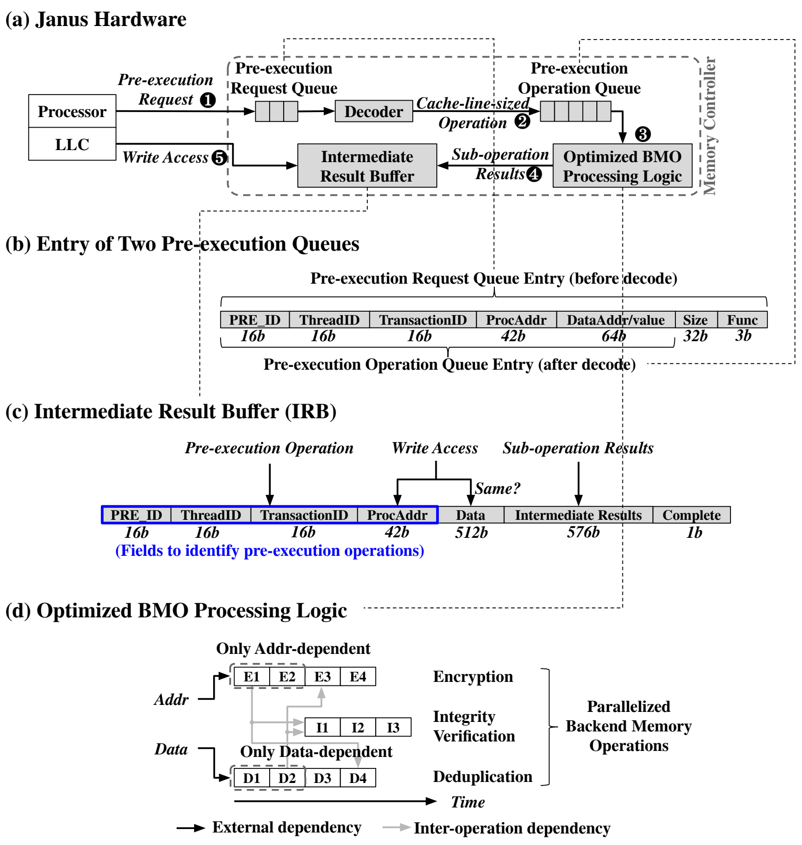

(a) Janus Hardware
Processor
LLC
Pre-execution Request ❶
Write Access ❺
Pre-execution Request Queue
Decoder
Cache-line-sized Operation ❷
Pre-execution Operation Queue
❸
Intermediate Result Buffer
Sub-operation Results ❹
Optimized BMO Processing Logic
Memory Controller
(b) Entry of Two Pre-execution Queues
Pre-execution Request Queue Entry (before decode)
PRE_ID 16b
ThreadID 16b
TransactionID 16b
ProcAddr 42b
DataAddr/value 64b
Size 32b
Func 3b
Pre-execution Operation Queue Entry (after decode)
(c) Intermediate Result Buffer (IRB)
Pre-execution Operation
Write Access
Sub-operation Results
Same?
PRE_ID 16b
ThreadID 16b
TransactionID 16b
ProcAddr 42b
Data 512b
Intermediate Results 576b
Complete 1b
(Fields to identify pre-execution operations)
(d) Optimized BMO Processing Logic
Only Addr-dependent
Addr
E1 E2 E3 E4
I1 I2 I3
Data
Only Data-dependent
D1 D2 D3 D4
Encryption
Integrity Verification
Deduplication
Parallelized Backend Memory Operations
Time
External dependency
Inter-operation dependency

Figure 4B.7: Detailed hardware mechanism of Janus.

than one transactions with the same `PRE_ID` to co-exist. Each IRB entry further contains another field, `TransactionID`, that distinguishes pre-execution requests across different transactions. These three fields (`PRE_ID`, `Thread_ID` and `Transaction_ID`) are assigned by the software interface which we will introduce in Section 4B.4.4. Using these fields, together with the physical address of the write (`ProcAddr`), Janus can uniquely identify pre-execution requests. Second, Janus needs to buffer pre-execution results for the actual write access when it arrives at the memory controller. The `IntermediateResults` field stores the intermediate results at cache line granularity. Considering the actual write access can arrive before the BMOs completes, a `complete` bit indicates whether all BMOs have completed or not.

Invalidates stale pre-execution results. The pre-execution becomes invalid if the memory or processor state it depends on has changed. Therefore, Janus invalidates the intermediate results from pre-execution by detecting any changes to the input memory or processor state. We summarize the potential cause of invalidation as the following two: (1) The input dependent data can be modified after the program issues the pre-execution request (e.g., due to cache line sharing, eviction or buggy

Table 4B.1: Software interface of Janus for pre-execution.

Type	Function	Description
Common	`PRE_INIT(pre_obj* obj)`	Initialize an pre_objwith a unique `PRE_ID`, the current `ThreadID` and `TransactionID`.
Immediate Execution	`PRE_BOTH(pre_obj* obj, void* addr, void* data, int size)`	Pre-execute all sub-operations.
	`PRE_ADDR(pre_obj* obj, void* addr, int size)`	Pre-execute address-dependent sub-operations.
	`PRE_DATA(pre_obj* obj, void* data, int size)`	Pre-execute data-dependent sub-operations.
	`PRE_BOTH_VAL(pre_obj* obj, void* addr, int data_val)`	Use an integer as the data. Pre-execute all sub-operations.
Deferred Execution	`PRE_BOTH_BUF(pre_obj* obj, void* addr, void* data, int size)`	Buffer pre-execution for all sub-operations.
	`PRE_ADDR_BUF(pre_obj* obj, void* addr, int size)`	Buffer pre-execution for address-dependent sub-operations.
	`PRE_DATA_BUF(pre_obj* obj, void* data, int size)`	Buffer pre-execution for data-dependent sub-operations.
	`PRE_START_BUF(pre_obj* obj)`	Start executing buffered pre-execution requests for pre_obj.

program that modifies the input data for pre-execution). In order to detect any stale value used in pre-execution, Janus keeps a copy of the data value that has been used for pre-execution in the `Data` field of IRB entry. When the write access arrives, IRB compares the data from the write access with this copy. If they are the same, the write access can safely use the intermediate results and complete the write to PM. Otherwise, data-dependent sub-operations have to be reprocessed using its new data. (2) Apart from the actual writes, pre-execution results buffered in the IRB may also depend on metadata structures employed by the BMOs. If these metadata structures are modified in such a way that they invalidate any prior pre-executed sub-operations, the pre-executed results must also be invalidated in the IRB. For example, pre-executing a deduplication sub-operation might identify that the current write (say to location B) is a duplicate of some prior write (say to location A). Therefore, the IRB stores the pre-execution result that the write to B is a duplicate of the value at A. However, before the pre-execution result is consumed, if an intervening write to location A changes the value of location A, then the pre-execution result in the IRB will be invalidated. In Janus, we extend BMOs to ensure that metadata changes trigger an IRB lookup and invalidates any stale pre-execution results.

Hardware Integration

Figure 4B.7a shows the detailed hardware mechanism. First, the processor sends the requests to a *Pre-execution Request Queue* that buffers the requests (step ❶). It supports two types of requests: (1) requests that start immediately, and (2) requests that are buffered in the queue and wait until the hardware receives an explicit start command. In the latter case, the requests with coalescing addresses will be merged within the queue for better efficiency (details in Section 4B.4.4). Second, a *decoder* decodes the request from the *Pre-execution Request Queue* into cache-line-sized operations and sends them to a *Pre-execution Operation Queue* (step ❷). Therefore, the pre-execution operations after the decoder stage all have one-cache-line granularity. Note that systems that perform BMOs at larger granularities (e.g., 256 B block for deduplication) can also be supported by modifying the decoder. Figure 4B.7b shows the fields in both queue entries. Third, the *Pre-execution Operation Queue* sends the decoded operations to the *Optimized BMO Processing Logic* (step ❸), and at the same time, it creates a new IRB entry. Figure 4B.7d shows the execution flow of the Optimized BMO Logic (correspond to the design in Figure 4B.6), where independent sub-operations can be executed in parallel and can be pre-executed if their external dependency is resolved. Finally, the *Optimized Backend Operation Logic* writes the pre-execution results to the previously created *Intermediate Result Buffer* entry (step ❹), which keeps track of the pre-execution at a cache line granularity, i.e., each entry in the buffer keeps the pre-execution result of one cache line. When the actual write access arrives, it can lookup the intermediate results from the IRB using its `ProcAddr` (step ❺). Note that the IRB, the *Pre-execution Request Queue*, and the *Pre-execution Operation Queue* have a fixed number of entries. If the buffer/queue is full, it drops newer requests. We discuss the software interface for our hardware mechanism in Section 4B.4.4, and discuss the system integration and exception handling in Section 4B.4.6. We present the hardware overhead in Section 4B.5.2.

Apart from the performance overhead, maintaining crash consistency is another issue as the BMOs have their own metadata. The *unreconstructable* metadata structures, the ones that cannot be rebuild using the data in PM, need to be kept up to date in PM when data gets updated. In the BMOs we have considered, there are three structures that cannot be reconstructed: counters for encryption, the deduplication address remapping table, and the root of the Merkle Tree. A recent work [28] has proposed counter-atomicity that atomically writes back both the encrypted data and its counter to PM in an encrypted PM system. In this work, we extend this atomicity to a more general metadata atomicity that writes back all unreconstructable metadata to PM atomically, ensuring that the processor can still read correct data during recovery. Note that the root of the Merkle Tree is typically protected by a non-volatile register in the secured processor [29, 57]. Therefore, it does not require any metadata atomicity. In order to reduce the atomicity overhead, Janus also follows the selective method on atomicity proposed by prior work [28], where only the writes that can immediately affect the crash consistency status (e.g., write that commits a transaction) requires metadata atomicity.

4B.4.4 Software Support for Optimization

Extensible. BMOs for PM can vary in different systems. A program developed with a software interface should be compatible with systems using different BMOs, without a need for additional software modifications. Therefore, Janus only exposes the two fundamental external dependencies to the software: the address and data of the write access. Table 4B.1 shows the software interface for pre-executing BMOs. Next, we explain how Janus provides an interface that can adapt to different PM programming models.

Programmable. Janus provides a structure, pre_obj, that has its unique `PRE_ID` and keeps track of the current `ThreadID` and `TransactionID`. These three elements enables the hardware to distinguish different pre-execution requests. To perform pre-execution on a object stored in PM, the programmer first needs to create a pre_obj and initialize it using `PRE_INIT`. Then, Janus provides two types of interfaces that enables programmers pre-execute the data structure that have either its address or data value available before the actual write to PM. Functions are identified by the field `Func` in the *Pre-execution Request Queue* entry (Figure 4B.7b).

The first type of function is for *immediate execution.* Janus provides three functions: `PRE_BOTH`, `PRE_ADDR` and `PRE_DATA`. Programmers can use them according to the availability of the dependent address or data. The input addresses are all virtual addresses from the program, and will be translated to processor-visible physical address (`ProcAddr`). Upon calling these functions, the pre-execution requests will be sent to the backend operation right away. Janus provides a special function, `PRE_VAL`, that takes a 64-bit integer value instead of the pointer to data. This function is designed to pre-execute transaction commit operations that typically set a valid bit or switches a pointer.

The second type is for *deferred execution.* Janus allows programs to *buffer* pre-execution requests using a class of functions that ends with the `BUF` suffix. These buffered requests can be executed together with the `PRE_START_BUF` function. Deferred execution provides more flexibility in scheduling the requests if the data structure to be pre-executed does not operate on a huge chunk of data, rather manipulates several elements in the structure separately. By buffering the requests for each element, the pre-execution buffer can merge the inputs before execution, leading to better efficiency.

Guideline for using the software interface. The hardware of Janus prevents misuse of the interface from causing any correctness issue. However, improperly placed Janus functions can lead to slowdown due to unused or discarded pre-execution. To effectively use the software interface, programmers need to be aware the following issues: (1) Between the pre-execution function call and the actual write operation, there should not be any update to the same location, or to the conflicting cache line. Although the underlying hardware mechanism can detect and fix such violations, the misuse can lead to a slowdown. (2) When using `PRE_DATA` alone, the data block must be cache-line-aligned (e.g., using alignment `malloc`). As the hardware tracks the pre_obj at cache line granularity, it is impossible to determine whether the data block shares its cache line with other data blocks without the address. Therefore, it is better to call pre-execution functions with `PRE_ADDR` or wait until both address and data become known if the programmer is not certain about the alignment. (3) As it takes a significant amount of time to execute the backend operations, it is better to place the pre-execution function calls sufficiently far away from the actual write. A simple and reasonable way to insert the pre-execution function call for a write request is to find the last update at that location and to insert that function right after that update.

Examples. Figure 4B.8a shows an example using the immediate-execution interface. First, we observe that the value used in the update operation (`val`) is available right after the function call (assuming nodes are cache-line-aligned). Therefore, a `PRE_DATA` function can be placed at line 4 to pre-execute the data-dependent BMOs. Then, we observe that the program uses an undo log to back up the node before modification (line 11). Therefore, it is possible to issue a pre-execution request for the address-dependent BMOs by inserting a `PRE_ADDR` function at line 8. Using these two pre-execution requests, we move the latency from BMOs off the critical path of the actual write (line 11). Figure 4B.8b shows an example of using the deferred-execution interface. The address and data for updates to `field1` and `field2` are already available after line 4. However, the two separate updates can be sharing a cache line (assuming the fields are not cache-line-aligned). The safe way to avoid invalidation of requests is to use the PRE_BUF function to buffer the pre-execution requests for each field and let them coalesce in the *Pre-execution Request Queue*. Then, placing a `PRE_START_BUF` function afterward will trigger the execution (line 10).

```
void updateTree(int key, item_t val) {
 pre_t pre_obj;
 // assume val is cache-line-aligned
 PRE_DATA(&pre_obj, &val,
                    sizeof(item_t));
 // find tree node with key
 node* location = find(key);
 PRE_ADDR(&pre_obj, location,
                    sizeof(item_t));
 // add old val to undo log
 undo_log(location);
 // update val
 location->val = val;
 // writeback updates
 clwb(&location->val, sizeof(item_t));
 sfence();
 ...
}
```

(a)

```
void updateTable(int id, item_t val1,
                      item_t val2) {
 // lookup entry location
 entry* location = tableLookup(id);
 pre_t pre_obj;
 PRE_BOTH_BUF(&pre_obj, &location->field1,
             &val1, sizeof(item_t));
 PRE_BOTH_BUF(&pre_obj, &location->field2,
             &val2, sizeof(item_t));
 PRE_START_BUF(&pre_obj);
 // backup old entry
 undo_log(location);
 // update fields
 location->field1 = val1;
 location->field2 = val2;
 // writeback updates
 clwb(location, sizeof(entry));
 sfence();
 ...
}
```

(b)

Figure 4B.8: Two PM transactions optimized by Janus.

4B.4.5 Compiler Support

The software interface of Janus is easy-to-use, but it still requires a good understanding of the program. To alleviate programmer's effort, we provide a compiler pass that automatically instruments the program with Janus functions.

Compiler Design

We develop our compiler pass on LLVM 7.0.0 [144]. The compiler pass analyzes and instruments the intermediate representation (IR) of the source code in the following steps. (1) The first step is to locate the blocking writeback operations (e.g., a `clwb()` followed by an `sfence()`), as these operations are responsible for moving the write latency on the critical path. (2) The next step is to perform a dependency analysis on the writeback objects. Our compiler pass takes two different analysis approaches for the data and the address of these objects. For address, it tracks dependencies of the address generation IR instructions of the object; for data, it tracks the modification to the memory address of the object. (3) The final step is to inject Janus functions (`PRE_DATA` and `PRE_ADDR`). The compiler pass injects them as far away from the actual writeback as possible in order to provide a better performance benefit. The injection approach is different for address and data. For address, it hoists the previously tracked dependent IR instructions for address generation to the beginning of the function, and places a `PRE_ADDR` function after the address generation is complete; for data, it places a `PRE_DATA` function between the last two updates on the object. It inserts the function as close as possible to the *pre-last* update using the data value assigned by the last update. Note that for both data and address, if the writeback operation depends on a conditional statement (e.g., if/else), our pass conservatively inserts the pre-execution function under the same conditional statement to avoid introducing potentially useless pre-execution requests. We evaluate our compiler pass in Section 4B.5.2 and compare it with our best-effort manual instrumentation.

Limitations

The compiler pass has the following limitations. First, it conservatively injects pre-execution functions within the same function as the writeback operation to guarantee correctness. Second, it can only inject pre-execution functions where both data and address dependencies can be resolved in compile time. For example, when a loop writes back an array of data, our pass cannot inject pre-execution for writebacks in the loop due to the lack of runtime information about the loop. Third, due to the lack of dynamic memory information, our compiler pass does not handle cache line sharing. We discuss future works that can mitigate these limitations in Section 4B.6.

4B.4.6 Real-World Considerations

This section described various scenarios that might arise while integrating Janus into real systems.

Unused pre-execution result. A buggy program can issue useless pre-execution requests without issuing a subsequent write access that uses the pre-executed result. Therefore, a useless pre-execution result can get stuck in the IRB. Janus takes a twofold approach to solve this problem: (1) Add an *age register* to each IRB entry, and discard an entry when the *age register* reaches its maximum lifetime. (2) Clear all entries belonging to a certain thread when that thread terminates.

Unused pre-execution request. A buggy program can also issue buffered pre-execution requests without starting their execution with a `PRE_START_BUF` function, causing congestion in the *Pre-execution Request Queue*. Janus solves this problem by using a fixed size FIFO for this queue. When the queue is full, it discards the *buffered* pre-execution requests at the top of the queue to make space for the new requests. Note that discarding pre-execution requests will never cause any correctness issue, but can result in missed opportunities to improve performance.

Memory swap. OS can swap memory to the disk and swap it back later. In this case, the physical address (`ProcAddr`) can be different. Our solution is to let the memory controller clear out all *Intermediate Result Buffer* entries that belong to the address range that will be swapped out.

Table 4B.2: System configuration.

Processor	Out-of-Order, 4 GHz
L1 D/I cache	64 kB/32 kB per core, private, 8-way
L2 cache	2 MB per core, shared, 8-way
Counter cache	512 kB per core, shared, 16-way
Merkle Tree cache	512 kB per core, shared, 16-way
Pre-exec. Request Queue	16 entries per core, shared
Pre-exec. Operation Queue	64 entries per core, shared
BMO Units	4 units per core (execute 4 BMOs in parallel), shared, perform at cache-line granularity
Intermediate Result Buffer	64 entries per core, shared
Memory	4 GB PCM, 533 MHz [28, 40, 197], $t_{RCD}/t_{CL}/t_{CWD}/t_{FAW}/t_{WTR}/t_{WR} =$ 48/15/13/50/7.5/300 ns [6]
Backend Operation Latency	AES-128 (Encryption): 40 ns [28, 48], SHA-1 (Integrity): 40 ns [48], MD5 (Deduplication): 321 ns [77]

4B.5 Evaluation

4B.5.1 Methodology

We model and evaluate a PM system that has three BMOs: encryption, integrity verification and deduplication (introduced in Section 4B.3.1 and 4B.4.3) using the cycle-accurate Gem5 simulator [205]. The system configuration is shown in Table 4B.2. The memory system is backed by Intel's ADR [43] support where all write accesses accepted by the write queue can drain to PM in case of a failure. The encryption and deduplication mechanisms follow a recent work [77], where the encryption counter and the deduplication address mapping table share the same metadata entry to minimize the storage overhead, i.e., if data is duplicated, the metadata entry stores the address mapping, otherwise, it stores the counter. The Merkle Tree is built on the co-located counter or deduplication address mapping to protect the integrity of both. We use selective counter-atomicity [28] to ensure crash consistency of counter-mode encryption, and extend this support to the other unreconstructable metadata, including the address remapping table in the deduplication mechanism and the message authentication code (MAC) in the integrity verification mechanism. We store the root of the Merkle Tree in a non-volatile register in the secured processor, as proposed in previous works [29, 57]. Similar to the ratio in prior deduplication works [76, 77], our main results use a deduplication ratio of 0.5. We also present the performance in other deduplication ratios in Section 4B.5.2. We evaluate and compare two system designs:

Table 4B.3: Evaluated workloads.

Workload	Description
Array Sway	Swap random items in an array
Queue	Randomly en/dequeue items to/from a queue
Hash Table	Insert random values to a hash table
RB-Tree	Insert random values to a red-black tree
B-Tree	Insert random values to a b-tree
TATP	Update random records in the TATP benchmark [217]
TPCC	Add new orders from the TPCC benchmark [218]

1. **Serialized:** Serialized backend operations.
2. **Janus:** Pre-execute the parallelized BMOs.

Our evaluation uses seven PM-optimized transactional workloads (listed in Table 4B.3), which are inspired by recent works [28, 40, 166], We evaluate the serialized design with the original program that only supports metadata atomicity. Then we manually instrument Janus primitives to evaluate Janus. We compare the manual and automated instrumentation through our compiler pass in Section 4B.5.2.

4B.5.2 Results

This section presents the results of our evaluation that compares the performance of the two design points. The workloads are single-threaded unless explicitly mentioned.

Single- and Multi-core Performance

In this experiment, we test the single- and multi-core performance of our design. Figure 4B.9 presents the speedup of Janus. Janus provides on average 2.35 $\sim$ 1.87× speedup in 1$\sim$8-core systems, respectively, over the serialized baseline system. We observe three broad trends from our results. (1) The speedup from pre-execution decreases as the number of cores increases because the memory bus contention increases when there are more threads executing in parallel, leading to a higher queuing latency in the memory controller. As a result, the ratio of BMO overhead decreases and the benefit of pre-executing BMOs also decreases. (2) The gain from pre-execution depends on the characteristics of the workloads: The speedup in B-Tree, TATP and TPCC is higher than that in Hash Table and RB-Tree. The reason is Hash Table and RB-Tree first look up the update location and then perform the update at that location. As a result, the address-dependent pre-execution request has a smaller window to execute and many times cannot complete before the

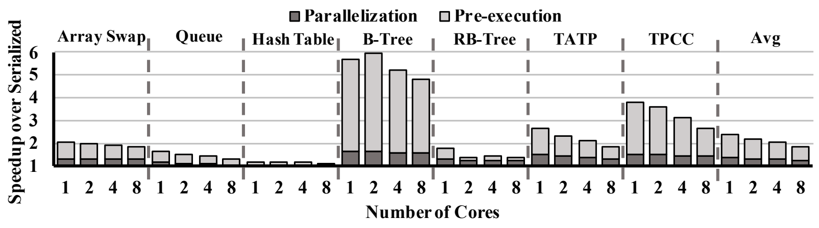


Figure 4B.9: Speed up of Janus over the serialized design with different number of cores.

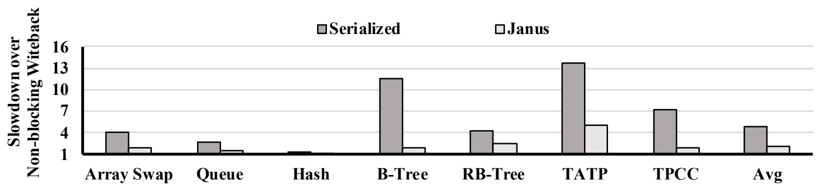


Figure 4B.10: Comparison with the ideal case where BMO latency is not on critical path.

actual write arrives. (3) Parallelization delivers a lower speedup compared to pre-execution because parallelization only reduces BMO latency, while pre-execution moves it off the critical path.

Comparison with Non-blocking Writeback

In this experiment, we evaluate an ideal case where the writeback requests do not block the execution. Therefore, the BMO latency is *not* on writes' critical path. We want to evaluate how much performance is lost when writes move on the critical path in crash consistent software and how much performance Janus can recover from that. Figure 4B.10 shows the slowdown of the serialized baseline and Janus over the ideal case. We observe that the serialized baseline introduces almost 4.93× slowdown when the BMO latency falls on the critical path. Janus improves the performance by 2.35× by pre-execution and parallelization of the BMOs. However, it still incurs a 2.09× slowdown compared to the ideal scenario. There are two reasons behind the performance gap between Janus and the ideal case. First, not all BMOs can be pre-executed, as sometimes there is not enough gap between the point where the data and address are known and where the actual write happens. Second, not all pre-execution requests can complete before the actual write access arrives. We found that in our experiments, on average, only 45.13% BMOs have been completely pre-executed.

Automated vs. Manual Instrumentation

In this experiment, we evaluate the performance of the automated instrumentation of Janus functions using our compiler pass. Figure 4B.11 shows the speedup of Janus with the *manual* and *automated* instrumentation over the serialized baseline. In most cases, the performance difference is within 12%. We notice two special cases. (1) The automated solution does not provide a significant performance

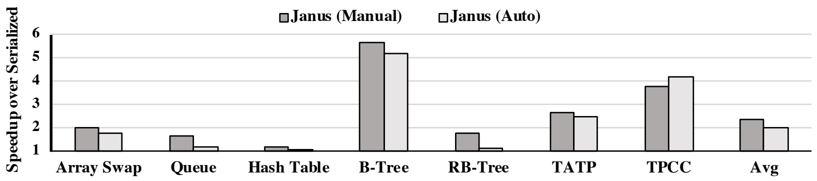


Figure 4B.11: Speed up of Janus over the serialized design with *automated* and *manual* instrumentation.

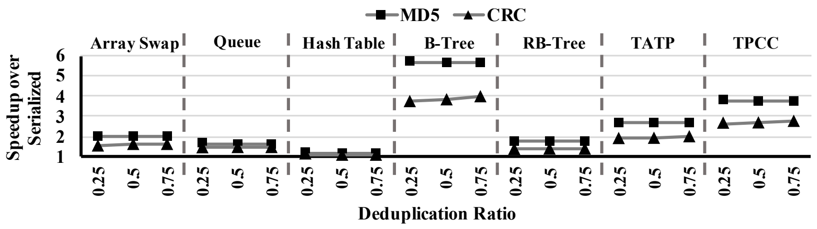


Figure 4B.12: Speedup of Janus over the serialized design with variable deduplication ratios and different algorithms.

benefit in RB-Tree and Queue. The static compiler cannot handle loops and pointers (discussed in Section 4B.4.5), which severely affects these two workloads. (2) The automated instrumentation is slightly faster in TPCC. We found that the instrumentation enabled other compiler optimizations on the program, such as hoisting the address generation. On average, the automated solution is only 13.3% slower than our best-effort manual instrumentation. We conclude that our compiler pass effectively finds opportunities for pre-execution and improves performance.

Different Deduplication Ratios and Algorithms

In this experiment, we test three deduplication ratios: 0.25, 0.5 and 0.75, and compare two different hashing algorithms: MD5 and CRC-32. The design using CRC-32 follows the method in [77], which has a lower overhead. Figure 4B.12 shows the speedup of Janus in systems using the MD5 and CRC-32 hashing algorithm. We observe that the speedup of Janus is almost the same under different deduplication ratios with MD5. In contrast, a higher deduplication ratio improves the benefit with the lightweight CRC-32. As MD5 takes around 4× longer than CRC-32, the BMOs dominate the write overhead. Therefore, the performance gain with MD5 is not impacted by the deduplication ratio. Even with CRC-32, the increase in speedup is small because BMOs contribute to most of the overhead, despite the benefit of deduplication.

Variable Transaction Sizes

In this experiment, we vary the size of the data update in each transaction from 64 B to 8 kB. As TATP and TPCC are real-world workloads that cannot be easily scaled without changing their semantics, we scale the first five workloads in this experiment. Figure 4B.13 shows the speedup

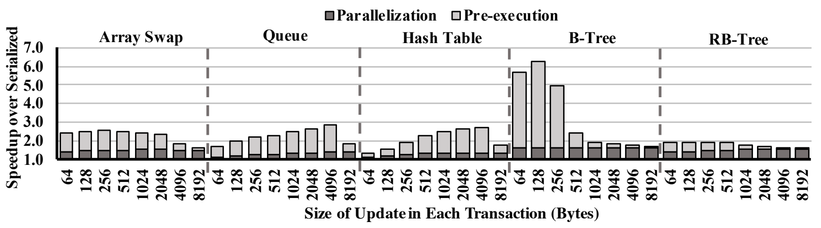


Figure 4B.13: Speedup of Janus over the serialized design with different transaction sizes.

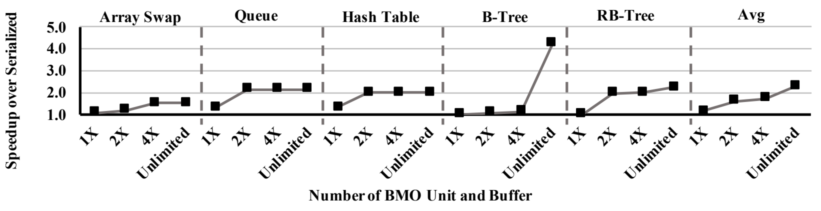


Figure 4B.14: Speedup of Janus over the serialized design with variable number of BMO units and buffer entries.

of Janus (parallelized and pre-executed) over the serialized baseline. We observe that the speedup from pre-execution increase with the size of transaction in the beginning, then it starts decreasing at a certain point in all workloads. In comparison to that, the speedup from parallelization keeps increasing but at a slow rate. The reasons are as follows: (1) Pre-execution benefits from a larger transaction size. However, at some point, the units and buffers for BMOs become full. The benefit is maximum at that point and then starts declining after that. (2) On the contrary, the benefit from parallelization is not affected by the BMOs resources. Therefore, the more writes the processor executes, the higher the benefit. We conclude that the speedup from pre-execution can benefit the performance the most when the write intensity is within a certain limit.

Variable Pre-execution Units and Buffer Size

The previous experiment has shown that the units and buffers for BMOs can become the bottleneck when processing large transactions. Therefore, in this experiment, we scale the number of units and buffers, while the size of transaction is fixed (8 kB) for each of the five scalable workloads. We test the speedup of Janus over the serialized baseline with 1×, 2× and 4× of the default number of units and buffers (listed in Table 4B.2). We also include a case with *unlimited* resources. Figure 4B.14 shows that as the BMOs units and buffer size increases, the performance also increases. However, the speedup in most cases saturates when the BMOs units and buffers are no longer the performance bottleneck. B-Tree is an exception. It exhibits a high demand for pre-execution resources and can gain a significant benefit with unlimited resources.

Overhead Analysis

Table 4B.2 in Section 4B.5.1 lists the size of buffers and queues that support pre-execution. The size of each Pre-execution Request Queue entry and Pre-execution Operation Queue entry is 119 bits and 103 bits, respectively. The size of each IRB entry is 148 B. In Janus, we have 16 Pre-execution Request Queue entries, 64 Pre-execution Operation Queue entries, and 64 IRB entries. Therefore, the total storage overhead from queues and buffers is 9.25 kB, which is 0.51% of the LLC size. The 4-wide BMOs in our design take 300 k gates (according to [219, 220]), which only incurs a 0.065 mm^2 die area with 14 nm technology.

4B.6 Future Works

This section discusses future works on memory and storage support for PM systems.

More precise compiler instrumentation. The limitation of our compiler pass boils down to the unavailable dynamic information during the static compilation time. There are two directions to mitigate this limitation. (1) Improving the dependency analysis on pointers can allow safe but more aggressive pre-execution. Techniques such as SVF [221, 222] can be greatly useful. (2) Utilizing dynamic analysis techniques can provide runtime information and enable more optimization opportunities, such as pre-executing BMOs outside of its function or outside its loop.

Tools for misuse detection. Section 4B.4.4 has described the guidelines on using Janus interface in order to gain the best performance. Future works can provide tools to detect misuse of the interface. There are three typical misuse scenarios: (1) *Modifications on pre-execution object.* Tools can detect whether the pre-executed address and/or data have been invalidated between the pre-execution function and the actual write. Address invalidation can be detected by monitoring memory de-allocation operations and data invalidations can be detected by monitoring assignments to the source of the data. (2) *Useless pre-execution functions.* Pre-execution on objects that do not affect the critical path is unnecessary. Tools can detect whether the pre-execution matches a subsequent blocking writeback. (3) *Insufficient pre-execution window.* The execution of BMOs takes a significant amount of time. The program should leave enough window between the pre-execution function and the actual write in order to maximize the benefit. A static tool can estimate the number of instructions in this window to determine whether the BMO latency can be perfectly overlapped; a dynamic tool can monitor the completion status of pre-execution functions and thereby adjust the instrumentation.

Chapter 5

Side-channel Attacks in Optane Persistent Memory

5.1 Introduction

Chapter 4B focuses on providing security guarantees for persistent data, and at the same time, achieving high performance. Instead of directly obtaining secret data, side-channel attack is another approach that indirectly infers information. Microarchitectural side-channel attacks use information from the microarchitecture layer to infer secrets on the software layer. Targets of side-channel attacks include hardware and software caches [103, 223–226] and branch predictors [227–229]. For example, Prime+Probe [223] can observe memory accesses at a cache set granularity, and Flush+Reload [103] further improves the granularity to a single cache line. Recently, transient-execution attacks [111], such as Spectre- and Meltdown-type attacks [23, 24, 112–116, 230], rely on side channels and have shown significant impact, drawing extensive attention. Especially in today's cloud environments, multiple users are co-located on the same server and share hardware components for better resource utilization [231]. Thus, side-channel attacks have become a prominent issue.

A successful microarchitectural side-channel attack requires detailed knowledge about the target microarchitecture. However, this knowledge, relevant for the security of the overall system, is usually not publicly documented but propriety. Therefore, prior works *reverse-engineered* hardware components to assess their relevance for security. For example, DRAMA [118] reverse-engineered the

DRAM addressing to establish a covert channel and spy on co-located processes; Gras et al., [119] reverse-engineered the translation-lookaside buffer (TLB) to leak sensitive information, such as cryptographic keys. Consequently, it is crucial to reverse-engineer *newer technologies* to assess their security properties before they are widely deployed and potentially threaten the users.

One such newer technology is a new type of memory, namely persistent memory. As Intel has released the Optane DC Persistent Memory (DCPMM) [232], this technology becomes commercially available.[1] Optane persistent memory DIMMs are installed on the memory bus alongside regular DRAM DIMMs, and deliver performance close to DRAM but persistence similar to hard drives. To leverage its high performance and persistence, systems usually expose Optane persistent memory directly to applications by mounting it in the direct access (DAX) mode (e.g., the EXT4 file system has a DAX mode optimized for Optane [233,234]). The DAX mode bypasses the file system, allowing programs to use load and store instructions to directly operate on persistent data. Therefore, Optane memory is good for storage-class applications, such as key-value stores [235–237] and databases [238–240]. As Amazon and Google offer Optane memory [241,242] already to cloud users, we need to ask the question: Does Optane persistent memory introduce new side-channel attacks that undermine system security and confidentiality?

In this chapter, we answer this question in the affirmative. We study and exploit side channels in the new Optane persistent memory. The foundation of our side-channel attacks is a thorough reverse-engineering of the microarchitectural (internal hardware) components of the Optane persistent memory. We identify and quantify correlations between memory access patterns and timing differences induced by Optane persistent memory. More concretely, we study the internal cache hierarchy and the controlling logic that prolongs the device lifetime via wear-leveling. As Optane is transparent to the processor via the DDR-T protocol [243], these elements of the Optane microarchitecture are not architecturally visible but only indirectly through timing differences.

Our reverse-engineering is the first to reveal security-critical low-level details of Optane's internal cache structures, E.g., their cache associativity and replacement policies, and the execution logic of the wear-leveling mechanism. Consequently, we construct four attack primitives for novel side-channel attacks and covert channels, based on (1) the Read-Modify-Write (RMW) buffer caching recently accessed cache lines in Optane, (2) the Address-Indirect-Translation (AIT) buffer caching recently used physical-to-internal address mappings, (3) read-write contention, inducing timing dif-

[1]Optane DC Persistent Memory is different from the older Optane-based NVMe SSD. For simplicity, we refer to Optane DC Persistent Memory as *Optane persistent memory* (or *Optane* in short) afterward.

ferences due to the conflicting concurrent operations, and (4) wear-leveling events, that induce latency-increasing effects.

In this chapter, we showcase four novel attacks using Optane persistent memory. First, we evaluate local cross-core covert channels based on our attack primitives, where the sender and receiver are co-located on the same server, sharing the same Optane DIMM. Even with isolation from the operating system, i.e., no direct data sharing and communication, the sender is able to transmit secrete data by creating timing differences via Optane internal structures. We evaluate three covert channels using attack primitives 1–3 described above.

Second, we present a keystroke timing attack, where a remote typer saves text into Intel's Optane-optimized key-value store, `pmemkv` [235,244,245]. A co-located attacker monitors the Optane DIMM's to observe events that update the typer's text in the key-value store. Thus, the attacker can record the inter-keystroke timing and potentially infer the typer's inputs.

Third, we present a remote covert channel, where sender and receiver run on different servers with network access to the `pmemkv` key-value store. The sender and receiver have a key that they can both update to communicate openly but want to exchange information covertly. That is, they do not exchange information directly using the values, i.e., the values can be completely unrelated text. We show that the high wear-leveling latency of the Optane memory is large enough (around 50 µs) for measurement across the network.

Finally, we present a remote *Note Board* attack, exploiting the persistence of Optane memory. Similar to the third attack, the sender and receiver are located on different servers, without direct message exchange. They do *not* probe the `pmemkv` server simultaneously. Instead, the sender stores a message on a covert *Note Board*, for the receiver to retrieve at a later time. This Note Board uses the internal properties of Optane behind a key-value store, by selectively applying repeated updates to different keys to set the wear-leveling metadata. As the wear-leveling metadata is persistent, even after 24 hours or reboots, the *Note Board* message can still be retrieved.

To summarize, this chapter makes the following contributions:

- We present the first side-channel security analysis of Intel Optane persistent memory, for which we reverse-engineer the cache hierarchy, cache sizes, associativity, replacement policies, read-write contention, and wear-leveling.

Table 5.1: System hardware and software configuration.

CPU	Intel Cascade Lake, 2.1GHz, 20 cores
DRAM	6x16GB DDR4, 2666MT/s
Optane	1x128GB Intel Optane DCPMM, App Direct mode, mounted as EXT4-DAX
NIC	Intel X550-T2, 10Gbps
Switch	Mikrotik CRS305-1G-4S, 10Gbps
Env.	Ubuntu 18.04, Linux kernel v5.4, gcc/++-7.5, PMDK v1.9, ndctl v68, ipmctl v02.00.00.3852

- We construct four attack primitives from our reverse-engineering, exploiting the timing of the RMW buffer, the AIT buffer, read-write contention, and wear-leveling.

- We demonstrate local and remote attacks, E.g., a remote keystroke timing attack on a remote typer, a remote covert channel where sender and receiver covertly communicate across the network, as well as local covert channels.

- We demonstrate a novel type of covert attack, exploiting the persistence property of wear-leveling in Optane memory. Our *Note Board* attack lets an attacker covertly store a secret message on a server using Optane, which a receiver can read even after 24 hours or a system reboot.

5.2 Reverse-engineering and Attack Primitives

Section 2.4.2 has shown that the Optane PM module is a complicated device, which is likely to be susceptible to side-channel attacks. In this section, we start with the foundation to the attack primitives—our reverse-engineering of low-level details of Optane. We then construct four new attack primitives based on the side channels in the different components.

5.2.1 System Configuration

Table 5.1 lists our system configuration: a Lenovo SR650 server with an Intel Xeon Cascade Lake CPU (20 cores) and an Optane DC Persistent Memory Module (DCPMM) installed alongside DRAM modules, running Ubuntu 18.04 (kernel v5.4). Optane runs in the App Direct mode for direct access using an Optane configuration tool, `ipmctl` [246]. The software system is configured with a compatible environment, including Intel's persistent memory controlling tool, `ndctl` [247], and a library for persistent memory, PMDK [248]. We mount Optane as EXT4-DAX [233, 234] for direct management of the persistent data, the typical setup of Optane [20, 248]. Through the paper,

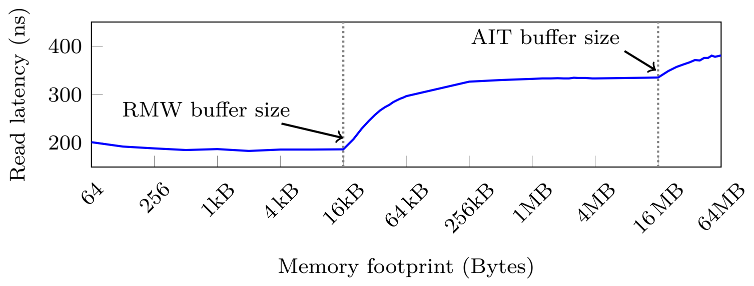


Figure 5.1: Optane read latency with variable memory sizes.

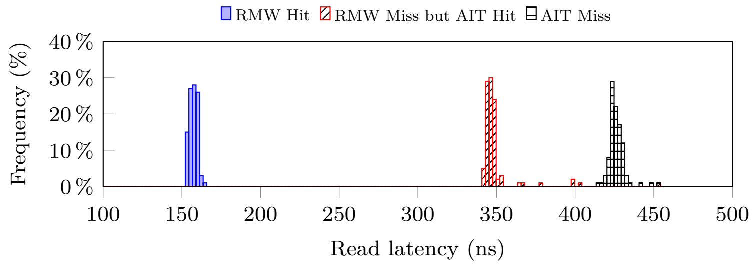


Figure 5.2: Hit and miss latencies of RMW and AIT buffers.

we follow this setup, with the exception of reverse-engineering (Section 5.2), where we disable the prefetcher to reduce the noise. In all case studies (Section 5.3–5.6), we enable all prefetchers to create a realistic environment.

5.2.2 Overall hierarchy in Optane

First, we reverse-engineer the internal cache hierarchy, i.e., the number of caches and cache sizes. We perform a unit test to find out the relationship between the memory footprint and the read latency. We take an approach similar to prior Optane characterization work [18]: the test program allocates variable-sized memory pools on Optane, and in each region, the program randomly reads 64 B chunks of data following a pointer-chasing pattern. As the program only accesses each 64 B chunk once, CPU caching does not affect the timing. Optane has large cache line sizes as discussed in Section 2.4.2 (256 B for RMW and 4 kB for AIT). Therefore, the first 64 B read brings data into the Optane-internal caches, and future accesses to adjacent 64 B blocks in the same Optane-internal cache line may become hits (if not evicted). This way, the footprint-latency relation can reveal Optane's cache sizes.

Figure 5.1 shows memory footprint (x-axis) and average read latency over 100 runs (y-axis). We observe two knee points, one at 16 kB and one at 16 MB. The first knee point is the Read-Modify-Write (RMW) buffer, and the second is the Address-Indirection-Translation (AIT) buffer. Figure 5.2

shows the distribution of AIT and RMW latencies. On average ($n = 100$), a read that hits RMW takes 157.3 ns ($\sigma = 1.5\,\%$), misses RMW but hits AIT takes 350.6 ns ($\sigma = 6.1\,\%$), and misses both RMW and AIT takes 426.5 ns ($\sigma = 1.2\,\%$). Note that the latency values in Figure 5.1 for RMW/AIT hits are higher because the first access to each RMW/AIT cache line is a miss but subsequent ones are hits. In summary, our results are consistent with prior works [18–21, 249].

We next focus our reverse-engineering on two internal cache structures (RMW buffer and AIT buffer) and two major effects (wear-leveling and internal read-write contention).

5.2.3 Read-Modify-Write Buffer

So far, we know the RMW buffer, the first caching structure an Optane access encounters, is 16 kB with a cache line size of 256 B. To enable side-channel attacks, there are two key properties: the cache replacement policy and its associativity. This section presents our reverse-engineering approach and our conclusions on these properties. In addition, we present our findings on instructions that can flush RMW entries.

Associativity

A set-associative cache usually determines the cache set using certain address bits—addresses that share certain common bits go to the same cache set. Inspired by prior work on CPU cache reverse-engineering [250], we take an approach that masks off different bits (i.e., set as the same value) and measure the Optane access latency. However, different from their approach, which directly uses performance counters to observe the latency, we measure the average access latency with a pointer-chasing approach similar to previous works on cache eviction [251, 252]. Specifically, our test program masks off bits from bit 8 (the bit after 256 B RMW cache line's block offset) to bit 21 (start counting from bit 0). In a set-associative cache, when bits that determine the cache set are masked off, the measured cache size is reduced, i.e., the knee point where read latency starts to increase comes early. We present the result in Figure 5.3a, where the x-axis is the memory footprint, the y-axis is the average read latency, and each legend indicates a curve with the labeled bit masked off (start counting from bit 0). Unlike a set-associative cache, we find that the measured RMW buffer size stays largely the same with different bitmasks. Note that we present five bitmasks for clarity; other bitmasks also have no latency effect. Thus, we conclude that the RMW buffer is fully associative.

Replacement Policy

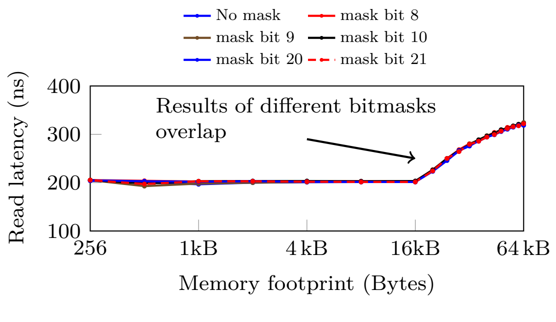
We reverse-engineer the replacement policy of the RMW buffer, i.e., in which order cache lines are replaced. We design a unit test that first fills up the RMW buffer with N distinct 256 B blocks, and then accesses them again in different orders: *same* order as first round, *reverse* order, and *random* order. According to prior works that reverse-engineers cache replacement policies [251, 253], an LRU cache has only hits in the second round if N is below the capacity, i.e., 64 for RMW, regardless of the access order. However, for $N > 64$, misses will happen. When accessing the same set of blocks in the same order as the first round, all reads are misses for an LRU cache, as the next read evicts the oldest line, which is exactly the next line to read. Figure 5.3b shows the RMW miss rate under these three access orders (100 runs each) and variable N values. Our result matches the behavior of an LRU cache, where the miss rate suddenly reaches 100 % when $N > 64$. In contrast, with the second reverse round, the first accesses still hit, which is better than the random access order. The random access order also has a higher miss rate than the reverse order. We conclude that the RMW buffer uses LRU replacement.

RMW Cache Flush

Though prior works have studied the caching effect in Optane [18–21, 249], there has not been any study on whether it is possible to flush data from the RMW buffer to gain direct access to the AIT buffer. We start with testing the `CLFLUSH` instruction. Figure 5.4 presents two histograms: one for the normal RMW hit latency and another for the case with a `CLFLUSH` to the whole 256 B RMW cache line between two reads. We observe that the normal RMW hit latency is 157.3 ns ($n = 100$, $\sigma = 1.5\,\%$); whereas with a `CLFLUSH` in between, the latency is 350.6 ns ($n = 100$, $\sigma = 6.2\,\%$), which is similar to an RMW miss. We also evaluated other cache flush/write-back instructions, `CLFLUSHOPT` and `CLWB`, and find that they both flush the RMW buffer.

Conclusion: The RMW buffer is a fully-associative cache with LRU replacement policy.[2] CPU instructions, such as `CLFLUSH` and `CLWB`, not only flushes CPU caches but also flushes RMW cache lines.

[2]Due to the high overhead of maintaining a true LRU policy, real-world processors tend to use pseudo LRU [254–256], which is also likely the case for Optane.



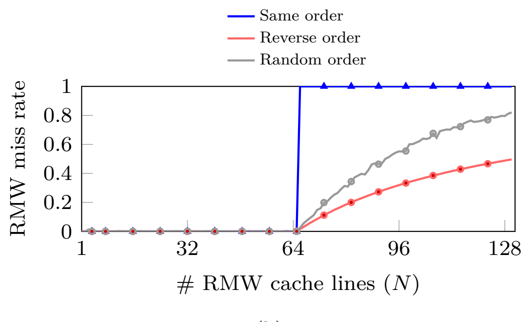


Figure 5.3: RMW (a) associativity using variable bitmasks and (b) replacement policy using different access patterns.

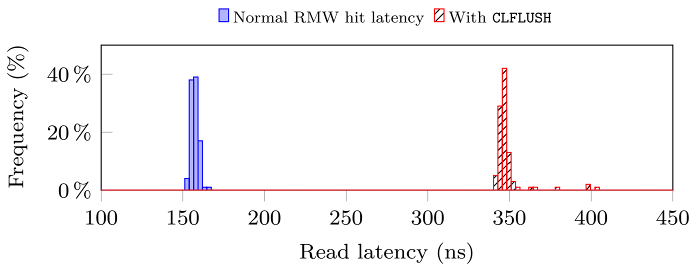


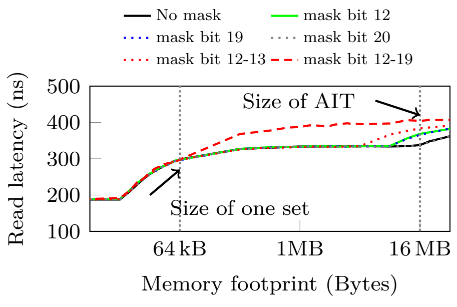
Figure 5.4: Effect of `CLFLUSH` to RMW buffer.

5.2.4 Address-Indirection-Translation Buffer

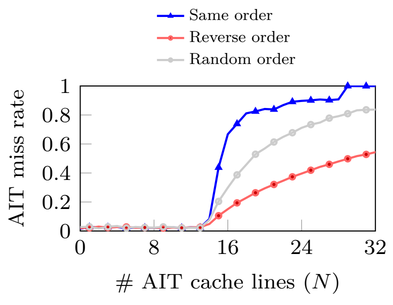
Optane has an internal address, different from the physical address, to enable wear-leveling and prolong the lifespan (Section 2.4.2). The AIT buffer caches the physical-to-internal mapping at 4 kB granularity, like the TLB in a CPU.

Associativity

Similar to reverse-engineering the RMW buffer, a unit test reading from different addresses determines whether the measured AIT capacity changes when masking address bits. We first mask bit 12 (after the block offset of 4 kB pages), and gradually increase the position of the masked-off bit. We measured the average latency over 100 runs with no bitmask (original latency) and all different bitmasks (Figure 5.5a shows 5 of them). We observe that the knee point, indicating the AIT buffer's capacity, shifts to 8 MB (1/2 of AIT capacity) when a bit between 12 and 19 is masked off but stops reducing when the bitmask moves to bit 20. We further mask off bit 12-13 and find the knee point becomes 4 MB (1/4 AIT capacity), and mask off all bits between 12-19 and observe a knee point of 64 kB (1/256 of AIT capacity). Thus, 64 kB is the capacity of one set. As each cache line in the AIT is 4 kB, one set contains 16 ways. Thus, the AIT is a 16-way set-associative cache, with bits 12-19 as index.



(a)



(b)

Figure 5.5: AIT (a) associativity using variable bitmasks and (b) replacement policy using different access patterns.

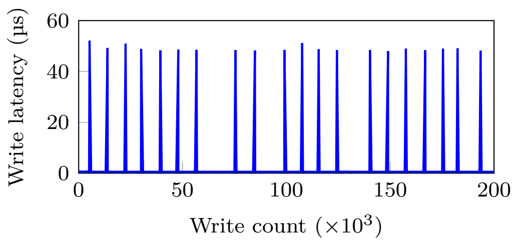
Replacement Policy

Like for the RMW buffer, we run a unit test reading a variable number of distinct AIT cache lines (4 kB) in three orders: same, reverse, and random. All AIT cache lines have the same bitmask (bit 12-19) to cache them in the same AIT set. To avoid the second round of accesses hitting the RMW buffer, we shift them by 256 B (i.e., original address + 256). Figure 5.5b presents the miss rate results (over 100 runs), as the number of AIT cache lines (N) and access order vary. Similar to the RMW results (Figure 5.3b), the miss rate increases when the number of AIT cache lines reaches 13. Prior work suggested that the AIT buffer may have a prefetcher [18]. Therefore, the miss rate may increase even before the size of each way (16). The same access order test has the worst miss rate increase, the reverse order performs best, with random order in between. Thus, same as in Section 5.2.3, we conclude that each set of the AIT buffer uses LRU replacement.

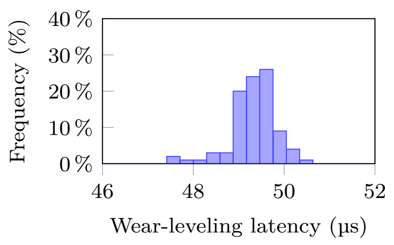
Conclusion: The AIT buffer is a 16-way set-associative cache (with 256 sets), with LRU replacement.^{2}

5.2.5 Wear-leveling

Wear-leveling in Optane remaps a physical address to a new page and migrates the existing data after this location has been repeatedly written to. Prior work on Optane memory has identified a significant latency increase after repeatedly writing 256 B of data to the same location [18] (finer-grained writes can be merged in the RMW buffer). We now perform a more thorough reverse-engineering of wear-leveling.



(a)



(b)

Figure 5.6: (a) latency of wear-leveling compared to normal writes and (b) a histogram of wear-leveling latency.

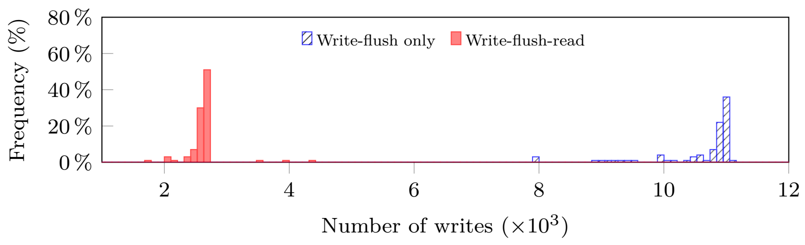


Figure 5.7: Number of writes to trigger one wear-leveling event.

Wear-leveling Timing

We first evaluate a unit test that repeatedly writes 256 B of data to the same location on Optane, similar to prior characterization work [18]. Figure 5.6a shows the write latency of a 256 B block (followed by a `CLFLUSH`) periodically increases. Figure 5.6b shows a latency histogram of 100 wear-leveling events: the average write latency is 562.8 ns ($n = 100$, $\sigma = 5.4\,\%$) but significantly increases to an average of 49.6 µs ($n = 100$, $\sigma = 2.5\,\%$) during wear-leveling. This observation is also consistent with prior work [18].

Effect of Reads and Writes to Wear-leveling

The wear-leveling latency is prominent but requires a large number of writes (>10 000) to trigger. Different from prior works that only perform writes [18], in this experiment, we test the effect of reads on wear-leveling counters. Figure 5.7 shows two histograms (both with 100 samples) on the number of writes it takes to trigger a wear-leveling event, one with writes and flush only (same as the experiment in Figure 5.6a), and another with a read to the same address after each write and flush (i.e., `Write+Flush+Read`). We observe that, with a read following each write, the average number of writes needed is 2625.7 ($n = 100$, $\sigma = 10.8\,\%$), compared to 11 646.8 ($n = 100$, $\sigma = 17.0\,\%$), if no reads. In addition, we test a case of read-only but fail to observe wear-leveling. Therefore, the read must be applied to a modified location to accelerate the wear-leveling effect.

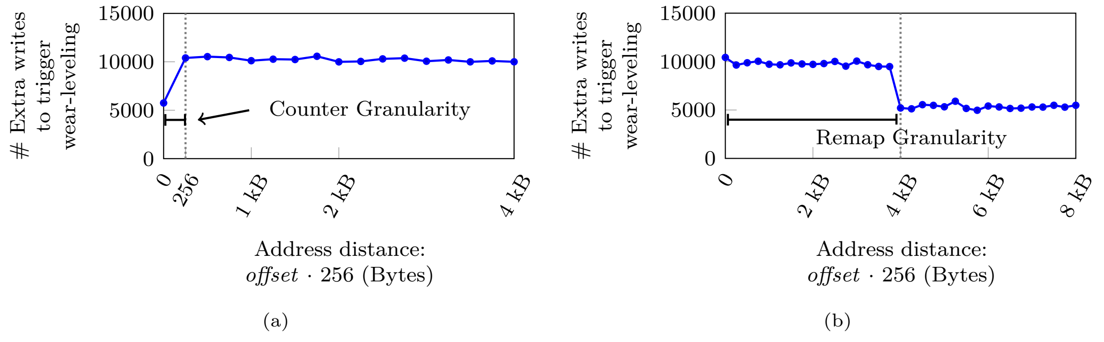


Figure 5.8: Experiments for reverse-engineering: (a) counter granularity and (b) remapping granularity of wear-leveling.

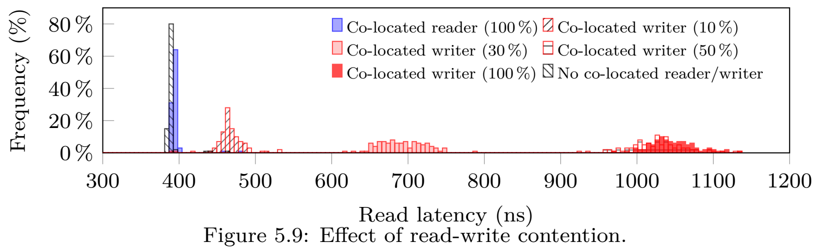


Figure 5.9: Effect of read-write contention.

Conclusion: The wear-leveling event has 88.1× higher latency than normal writes. With a read following each write (same location), the number of writes needed to trigger wear-leveling can be 4.4× less. This finding makes it more practical to construct a wear-leveling-based channel.

Wear-leveling Granularity

Though prior characterization [18] has shown that the internal- to physical-address mapping has a granularity of 4 kB, the wear-leveling granularity remains unknown. To use wear-leveling as an attack primitive, we target two new research questions. (1) As the wear-leveling counter determines whether a block needs to be remapped, what granularity does each wear-leveling counter cover? (2) When a remapping happens, what is the granularity of remapping?

(1) Counter granularity. We take a novel approach that initializes the wear-leveling counter of a 256 B block and then checks how many extra writes it takes to trigger a wear-leveling event on top of the initialized wear-leveling counter in nearby locations. This approach can determine the granularity each wear-leveling counter covers. Based on this idea, the test program first performs an initial of 5000 writes of 256 B blocks to a 4 kB-aligned location A (with a flush after each write). Then, it measures the number of additional 256 B writes it takes to trigger a wear-leveling event at location $A + \mathit{offset} \cdot 256$. Figure 5.8a shows this number with variable *offset* values (average over 100 runs). We observe that, when $\mathit{offset} \cdot 256 = 0$, the number of additional writes is around 5000.

However, when $\mathit{offset} \cdot 256 > 0$, the additional writes are always greater than 10 000, indicating that the initial 5000 writes have not been taken into the counter. Therefore, the wear-leveling mechanism has a counter granularity of 256 B.

(2) Remapping granularity. We take another novel approach that initializes wear-leveling counters of a 256 B block (*i.e.,* location A), trigger wear-leveling at a nearby location $A + \mathit{offset} \cdot 256$, and measure how many extra writes it takes to trigger wear-leveling at A. If the other location $A + \mathit{offset} \cdot 256$ falls into the remapping granularity with A, it will take more writes to trigger wear-leveling at A again as remapping has happened; otherwise, it will take less writes as the initialization of wear-leveling counters at A remains. Specifically, the test program first performs 5000 initial writes of 256 B blocks to a 4 kB-aligned location A. Then, it keeps writing to location $A + \mathit{offset} \cdot 256$ until wear-leveling is detected. Finally, it measures the number of writes to trigger wear-leveling at location A. Figure 5.8b shows this number with variable *offset* values ($n = 100$). We see that when $\mathit{offset} \cdot 256 < 4kB$, the average number of extra writes to trigger a wear-leveling is around 10 000 but halved when $\mathit{offset} \cdot 256 \geq 4kB$. Thus, the remapping granularity is 4 kB, matching the AIT granularity.

Conclusion: The wear-leveling mechanism has a remapping granularity of 4 kB but each individual 256 B block has its own counter for wear-leveling. Once wear-leveling happens, the counters in all 256 B blocks are reset.

5.2.6 Read-Write Contention

Past characterizations on Optane [19, 20, 257] reveal that the read bandwidth of Optane is around twice higher than that of writes, but they do not study how writes affect the timing of reads. To understand read-write contention in Optane, we design a unit test program, where a main thread performs random reads using pointer-chasing and another co-located thread performs random reads/writes at the same time (to independent addresses). Both threads are pinned to different cores using `taskset`. We further control the type of accesses in the other thread as well as the intensity. Figure 5.9 demonstrates six histograms ($n = 100$ in each) for the read latency of the main thread, with different types of co-located threads: 100 % read intensity and 30–100 % write intensity. We observe that, with another reader thread of 100 % intensity, the main thread has a minor increase in latency—389.5 ns ($n = 100$, $\sigma = 2.8\,\%$) of normal read latency increased to 395.1 ns ($n = 100$, $\sigma = 2.8\,\%$). In comparison, with another writer thread, even at 10 % intensity, the increase in latency is significant (average is 466.9 ns, $n = 100$, $\sigma = 4.1\,\%$). And, with higher write intensity (100 %) in the co-located thread, the read latency increases to 1047.9 ns ($n = 100$, $\sigma = 2.9\,\%$).

Conclusion: In Optane persistent memory, writes can seriously content with reads and cause read latency to increase. Therefore, it is possible to sense write activities from other programs using a unit test of reads.

5.2.7 Summary of Attack Primitives

In summary, we build four attack primitives using the following timing channels:

- There is an exploitable difference of 193.3 ns between hit and miss latency of the RMW buffer during read access.
- There is an exploitable difference of 75.9 ns between hit and miss latency of the AIT buffer during read access.
- For read-write contention, the read latency has a significant increase of 658.4 ns with background write activities.
- A higher wear-leveling latency due to repeated writes—an increase of 49.0 µs latency over normal writes.

5.3 Local Cross-Core Covert Channel

In this case study, we evaluate local cross-core covert channels based on our attack primitives. The transmission rates are upper bounds for the capacity of our side channels, following the methodology of prior works [103–105, 258–260].

5.3.1 Attack Model

We assume that sender and receiver are co-located on a server, using different cores, and share the same Optane DIMM. They are isolated by the OS without any means to communicate. Sender and receiver maintain separate memory-mapped files on an Optane DIMM, isolated by the file system. The platform follows the same configuration as Section 5.2.1, with CPU prefetchers enabled. We illustrate this setup in Figure 5.10.

5.3.2 Attack Design

To establish a covert channel, we use three attack primitives: the timing differences of the RMW buffer, the AIT buffer, and read-write contention. Next, we explain the details.

RMW-based covert channel. As the RMW buffer is a cache-like structure, we take the commonly-used Prime+Probe approach to establish the covert channel. The sender reads from the sender's file repeatedly when sending a bit of 1, and stays idle when sending a bit of 0 (step ❶). In the meantime, the receiver keeps performing Prime+Probe (step ❷): first read from a set of random locations (in a pointer-chasing pattern) on the receiver's memory-mapped file (as prime), wait for the sender's activities, and read from these locations again (as probe). Thus, sender's reads will evict receiver's reads from RMW and increase the probe latency. However, reads may hit CPU caches before accessing the RMW buffer in Optane. Therefore, we take advantage of the larger cache line size of the RMW buffer by shifting the accesses by 64 B during probe. This way, the reaccesses during *probe* can bypass CPU caches and check if the primed locations hit the RMW buffer.

AIT-based covert channel. Due to the similarities between the AIT and RMW buffers, we take a similar Prime+Probe approach as the RMW buffer, except for two differences. First, besides the CPU cache, a channel based on the AIT buffer also needs to avoid RMW cache hits. Based on the reverse-engineering on RMW buffer flush (Section 5.2.3), the receiver's program issues `CLFLUSH` instructions to locations covered by the accesses during *prime*. Thus, accesses during *probe* can bypass the RMW buffer and infer whether these locations hit the AIT buffer. Second, as the AIT buffer is set-associative (Section 5.2.4), both the sender and the receiver only read from addresses that belong to the same AIT set.

Read-write-contention-based covert channel. The sender writes to the sender's file when sending a bit of value 1 and stays idle when sending a bit value of 0 (step ❶). In parallel, the receiver performs random reads (step ❷) following a pointer chasing pattern to create cache misses (CPU caches, RMW buffer, and AIT buffer) and fetch data from the Optane media. As our reverse-engineering in Section 5.2.6 has shown, the existence of writes can significantly degrade the read latency. Thus, when detecting a significant increase in read latency, the receiver can determine that the current bit is a 1.

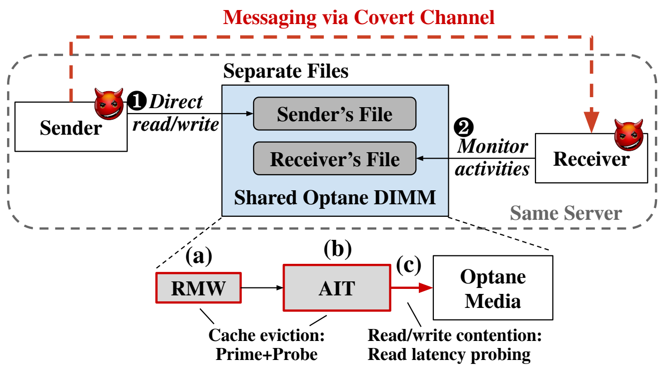


Figure 5.10: Local covert channels based on the (a) RMW buffer, (b) AIT buffer, and (c) read-write contention.

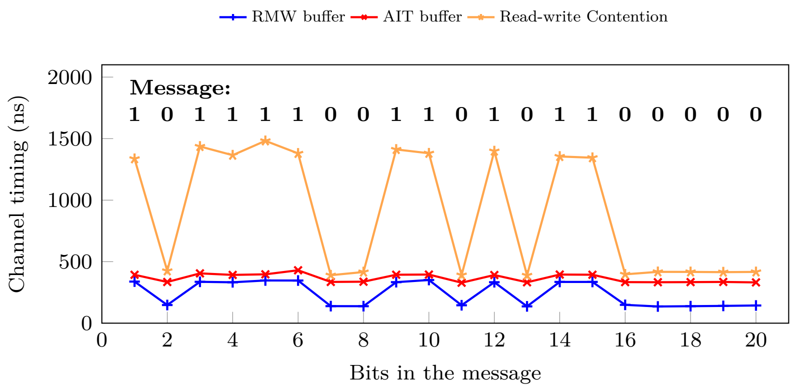


Figure 5.11: A demonstration for the local covert channels.

5.3.3 Attack Setup

We run these local covert channels in a system described in Section 5.2.1. The sender and the receiver create two separate files on a shared Optane DIMM, following our attack model (Section 5.3.1). Because of the LRU replacement policy (Figure 5.3b and 5.5b), we find that the receiver only needs to prime a few buffer entries, as long as the sender causes sufficient evictions. In the RMW-based covert channel, the receiver primes 8 entries and probes them with a fixed threshold of 238 ns; in the AIT-based covert channel, the receiver primes 3 entries and probes them with a fixed threshold of 376 ns.

5.3.4 Results

Demonstration. Figure 5.11 demonstrates our approach. The x-axis shows the bit sequence in the message and the y-axis shows the timing differences the receiver observes in each channel. When sending a bit value of 1, the receiver's prober can detect a latency increase. We observe that the read-write contention has the most significant effect, which is consistent with our finding that writes

Table 5.2: Local covert channel ($n = 100$).

Channel	BW ($kbit/s$)	Acc (%)	σ_{BW}	σ_{Acc}
RMW	11.35	99.60	0.005 %	0.17 %
AIT	10.50	98.26	0.004 %	1.81%
Contention	2.33	99.60	0.0003 %	0.14%

Table 5.3: Comparison with existing cross-core covert channels without shared memory.

Methods	Bandwidth	Error Rate
DRAMA [118]	300 kB/s	1.8 %
Prime+Probe [260]	67 kB/s	0.36 %
Memory Bus Locking [261]	93 B/s	0.09 %
RAPL [264]	2.3 B/s	0.89 %
This work (with RMW buffer)	1.42 kB/s	0.4 %

can seriously content with reads (Section 5.2.6). In comparison, the hit/miss timing difference in RMW is less substantial, and is the lowest in AIT.

Bandwidth and accuracy. We evaluate each channel by having the sender transmit 1000 bits to the receiver over 100 runs. Table 5.2 presents the results. We observe that the RMW- and AIT-based channels have similar bandwidth, 11.35 and 10.50 kbit/s, but the contention-based channel has a lower bandwidth of 2.33 kbit/s. Although the timing difference from read-write contention is significant, the sender needs to spend more time performing writes due to the slower write performance. Despite the bandwidth differences, all three covert channels have accuracy higher than 98 %.

Comparison with existing covert channels. Our cross-core covert channel does not rely on a shared memory. Our results are in a similar range as other covert channels that do not rely on shared memory [101, 102, 118, 261–264]. Compared to other cache-based covert channels [101, 102, 259], similar techniques can be applied to improve the performance of the covert channel. The covert channel noise can be further reduced by applying more advanced statistical and error-correction techniques (e.g., the proposal by Maurice et al., [259]). Table 5.3 compares our Optane-based cross-core covert channel with existing methods.

5.4 Keystroke Attack

In this section, we introduce a case study of the *keystroke side-channel attack* using Prime+Probe on the RMW buffer.

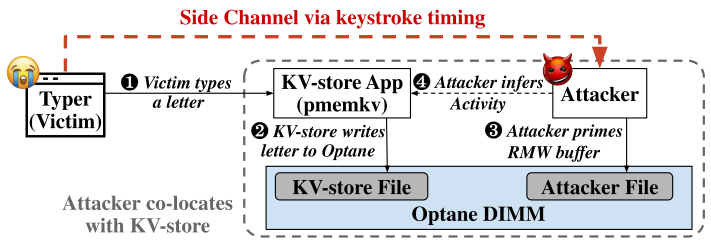

Side Channel via keystroke timing
Typer (Victim)
❶ Victim types a letter
KV-store App (pmemkv)
❹ Attacker infers Activity
Attacker
❷ KV-store writes letter to Optane
❸ Attacker primes RMW buffer
KV-store File
Attacker File
Optane DIMM
Attacker co-locates with KV-store

Figure 5.12: Keystroke side-channel attack.

5.4.1 Attack Model

We assume a scenario where a victim types into a web interface, and each keystroke is sent to a web server that maintains storage on Optane. For every keystroke typed by the victim, the website sends an update request to the key-value store (KV-store) server in order to track the user's latest update. We assume that the attacker is co-located with the KV-store application in the same server and shares the same Optane DIMM. However, the existing OS-level isolation disallows any direct communication between the attacker and the KV-store.

5.4.2 Attack Design

Similar to a Prime+Probe attack, the attacker can infer keystrokes via the RMW side channel as follows. The keystroke side-channel attack assumes an attack model as described in Figure 5.12. First, the attacker types a letter which is transmitted via WebSockets to the KV-store server (step ❶). The KV-store then stores the letter to the Optane memory (step ❷). In parallel, the attacker constantly primes the RMW buffer (step ❸) and then probes the memory by reaccessing. The attacker infers whether a key is inserted based on the timing of reaccesses (step ❹). The Prime+Probe approach is similar to the one in the RMW-based local covert channel (Section 5.3.2). When a key was inserted due to the typer's keystroke, the attacker can sense an increased latency in the probing step; when the timing stays low, the attacker can deduce with a high probability that no data was inserted into the KV-store, i.e., no keystroke activities.

5.4.3 Attack Setup

We run the experiment in our lab environment using two servers connected via a hardware switch in the local network (configuration in Table 5.1). We choose Intel's Optane-optimized KV-store `pmemkv` [235, 244, 245] as the backend storage, which is then connected through WebSocket to save the typer's inputs. We use a public keystroke dataset containing inter-keystroke latencies from 100 different typers typing the same eight-letter password "try4-mbs" 10 times [265], resulting in a

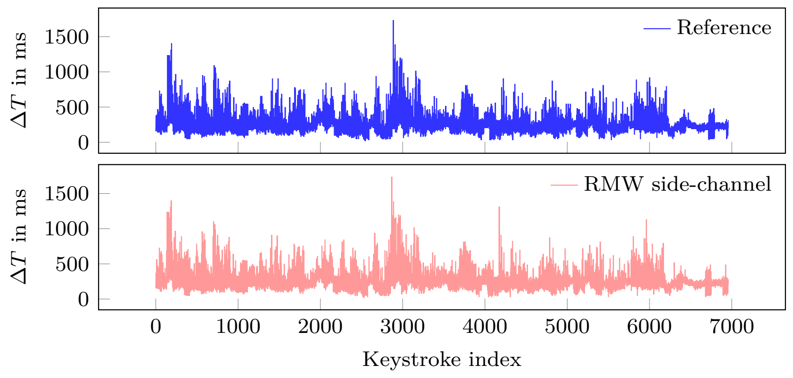


Figure 5.13: The inter-keystroke timings (ΔT) from the typer (top) and the RMW side-channel (bottom).

total of 7000 inter-keystroke timings. The client (victim) simulates the individual typers by sending keystrokes delayed by the prerecorded inter-keystroke timings. As Figure 5.2 shows, an RMW hit can be clearly distinguished from a miss. Thus, we choose a fixed threshold of 285 ns to distinguish RMW hits from misses. The attacker starts the Prime+Probe attack with a co-located program and detects the inter-keystroke timings, by *probing* the RMW buffer every 9.52 ms.

Our evaluation covers a noise-free scenario and scenarios with other co-located activities. We run *another* `pmemkv` instance that shares the same Optane DIMM, which continuously processes random, independent requests. As writes to Optane have higher latency impact (Section 5.2.6) and can trigger wear-leveling (Section 5.2.5), we take a relatively update-heavy input [266,267], which consist of 80 % read (`GET`) and 20 % update (`PUT`) requests. Under this ratio, we evaluate three levels of request intensity: 70 %, 40 %, and 10 %, corresponding to *High*, *Medium*, and *Low* background noise levels.

5.4.4 Results

To determine the accuracy of our attack, we calculate the timing difference between the ground-truth latencies from the prerecorded dataset and the detected ones. We repeat the experiment 100 times with all the 7000 inter-keystroke timings and observed an overall error rate of 1.04 % in the no-noise scenario. Figure 5.13 shows the results of one run in the time domain, where the difference between the RMW side-channel and the ground-truth is negligible. Figure 5.14 shows further analysis of the error distribution of the RMW channel compared to the timing distributions of the ground-truth. The distribution of the ground-truth inter-keystroke timings (on average 271.90 ms, $\sigma = 53.47\,\%$) is 82.4× larger compared to the error of the received timings over the RMW channel (on average 3.30 ms, $\sigma = 68.78\,\%$). We observe a maximum time difference between the RMW channel and the

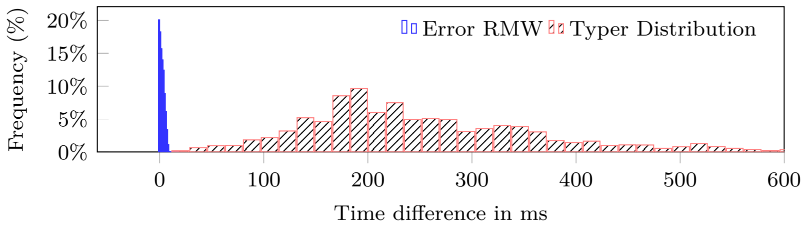


Figure 5.14: The time distribution of the reference typers compared to the error's distribution of the RMW side-channel.

Table 5.4: Error rates of the keystroke side-channel.

Noise	Error (%)	σ
No	1.04	0.26 %
Low	28.66	16.54 %
Med	88.95	2.72 %
High	100.00	0.00 %

ground truth of about 20 ms. The error rate of 1.04 % consists of two distinct error types. First, the inter-keystroke timing can be split into two RMW events, leading to smaller observed differences in the RMW side-channel. Second, two inter-keystroke timings can be combined into a single RMW event, leading to a larger observed time difference on the attacker end. In a real world attack, the inter-keystroke timings of a user are typically independent from the previous keystroke timings, leading to only one miss predicted keystroke. In some cases, the *event splitting* can also be corrected when considering the probability of a given timing difference.

We also evaluate the impact of three different levels of background noise (see Table 5.4). Under low-noise, the error rate is 28.66%. However, under higher noise levels, the side channel degrades to 100% error under high-noise. This is in line with prior work on keystroke side-channel mitigation [268], i.e., a low-frequency event like a keystroke is easily buried in a large amount of noise.

Comparison with existing keystroke attacks. Inter-keystroke timing have become a popular showcase for software-based side-channel attacks. Some operating system interfaces allow observing or inferring keystroke timings [269, 270]. Side-channel attacks exploited CPU usage [271], CPU caches [104, 260] with Flush+Reload, CPU caches with Prime+Probe on L1 [272] and on L3 [268]. Crucial to all these attacks is a highly precise measurement of the keystroke timestamp. We note that our attack is on par with the state-of-the-art, enabling the same end-to-end attacks. However, these previous attacks have been local attacks, whereas ours works in a remote scenario. Two previous works also explored the remote keystroke-timing scenario [110, 273]. Song et al., [273] mounted a timing attack on packets sent over an SSH connection. While they also attack keyboard input of a

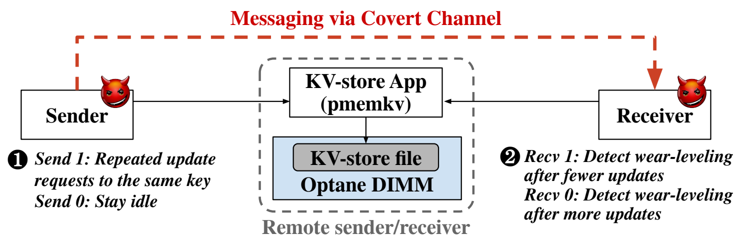

Messaging via Covert Channel
Sender
KV-store App (pmemkv)
KV-store file
Optane DIMM
Receiver
❶ Send 1: Repeated update requests to the same key
Send 0: Stay idle
❷ Recv 1: Detect wear-leveling after fewer updates
Recv 0: Detect wear-leveling after more updates
Remote sender/receiver

Figure 5.15: Remote covert channel.

remote user, they only provide quantitative data for the end-to-end password recovery but not for the channel itself. Kurth et al., [110] mounted a remote keystroke-timing attack, on DDIO via RDMA. While the experimental setup is slightly different, they also try to recover millisecond-accurate inter-keystroke timings of a remote user. In a scenario without noise, they achieve an F-Score 0.66. For comparison, our attack achieves an F-Score of 0.99 in the no-noise scenario.

Each inter-keystroke timing is statistically independent and our evaluation focuses on the mean timing difference of the inter-keystroke timings compared to the ground truth. To infer written language or guess passwords more advanced techniques such as machine learning can be applied [265, 269, 273–276].

5.5 Remote Covert Channel

In this section, we introduce the third case study on a *remote* wear-leveling-based covert channel.

5.5.1 Attack Model

We assume the same scenario as in Figure 5.15, where sender and receiver are located on different servers but have access to another KV-store server through the network (one-hop via a switch as listed in Table 5.1); the sender and the receiver do not have a direct method of communication. In the KV-store, they have access to common keys. However, the sender and the receiver stay stealthy, without sending any direct messages via the KV-store. An example of such a shared KV-store can be an online document that different users can update.

5.5.2 Attack Design

To communicate with the client, our server implementation uses the IPv4 protocol with TCP sockets (`SOCK_STREAM,AF_INET`). As the number of writes to trigger a wear-leveling is stable (Section 5.2.5),

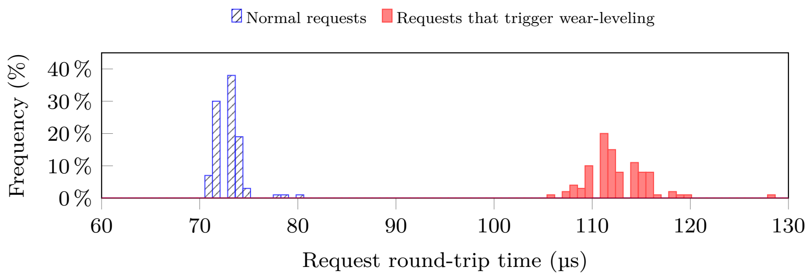


Figure 5.16: Histogram of the remote request RTT ($n = 100$).

our high-leveling idea is to have the sender help trigger a wear-leveling event when sending a bit value of 1. The sender continuously sends update requests to the KV-store server when sending a bit value of 1, and stays idle when setting a bit value of 0 (step ❶). Correspondingly, the receiver also sends repeated update requests to the server, and at the same time, count the number of update requests to trigger a wear-leveling event (step ❷). When the sender is transmitting a bit value 1, the receiver needs fewer requests to observe wear-leveling latency as compared to a bit value 0.

Challenges. However, there are two major challenges that can degrade the channel. (1) Requests sent through the network are not as intensive as local experiments during the reverse-engineering. If the time gap between two requests is large, the receiver cannot easily distinguish the wear-leveling latency through the network as part of the wear-leveling latency has been overlapped with this time gap. (2) To trigger wear-leveling, the writes need to update an entire, aligned 256 B block (Section 5.2.5). However, the allocation within the KV-store program does not guarantee 256 B alignment.

Solutions. We take two approaches to overcome these practical challenges. First, instead of one thread, the sender issues four threads to send update requests to mitigate the time gap that may overlap with the wear-leveling latency. Second, the sender updates a 512 B block in the request (as value). This way, the update at least covers one 256 B block.

5.5.3 Attack Setup

The hardware platform follows the configuration in Table 5.1 (the CPU has prefetchers enabled). On the software side, the server runs Intel's `pmemkv`, a key-value store optimized for Optane [235]. The `pmemkv` interface takes both `PUT` and `GET` requests. Notably, for a `PUT` request, if the key already exists and the size of value remains the same, `pmemkv` updates the value directly in-place; otherwise, it creates a new key-value entry. During an update, `pmemkv` *reads* the existing value and makes a

Table 5.5: Remote covert channel under different levels of background noise ($n = 100$).

Noise	BW (bit/s)	Acc (%)	#Pkt/bit	σ_{BW}	σ_{Acc}	$\sigma_{\#Pkt/bit}$
No	10.01	98.87	2794.83	0.29 %	1.05 %	1.14 %
Low	10.01	90.00	2789.54	0.29 %	3.38 %	1.03 %
Med	10.00	88.57	2790.91	0.19 %	3.14 %	1.01 %
High	10.01	88.40	2781.18	0.30 %	3.02 %	1.16 %

Table 5.6: Comparison with existing remote covert channels (local network, without background noise).

Methods	Bandwidth	Error Rate
DDIO [110]	16 kbit/s	0.2 %
This work (with wear-leveling)	10.01 bit/s	1.13 %
NetSpectre [117]	1.07 bit/s	<0.1 %
Memory Deduplication [277]	0.08 bit/s	0.6 %
FS Deduplication [278]	0.05 bit/s	2.5 %

backup to maintain data recoverability. This procedure helps accelerate wear-leveling, according to our reverse-engineering in Section 5.2.5. Similar to the keystroke attack, we also consider scenarios that are noise-free and those with co-located activities (methodology in Section 5.4.3). In this experiment, the sender transmits a 100-bit message to the receiver. And, we repeat this experiment for 100 times.

5.5.4 Results

Bandwidth and accuracy. Figure 5.16 compares the round-trip time (RTT) between normal requests and requests that trigger wear-leveling. On average, normal requests take 72.99 µs (σ = 1.93 %) and those with wear-leveling take 113.54 µs (σ = 5.21 %). Therefore, even with one event of wear-leveling, the request timings are already distinguishable. Table 5.5 presents our results under different background noise levels, as well as a basic scenario that does not have a co-located pmemkv running in the background. Among the four scenarios, the bandwidth values are close (around 10 bit/s) and the accuracy values remain good even with high background noise. As the wear-leveling event has a latency of 49.6 µs, significant compared to the tens-of-µs network latency, it is not surprising that this channel is robust and stable. Further, as each update request (one packet per request) contains both write and read (Section 5.5.3), the number of packets needed to trigger a wear-leveling event (i.e., one bit in the message) is lower than pure writes (around 2800), which is consistent with our observation in Section 5.2.5.

Comparison with existing remote covert channels. Table 5.6 compares our work with several prior works on remote covert channel. Our work achieves a higher bandwidth than NetSpectre [117]

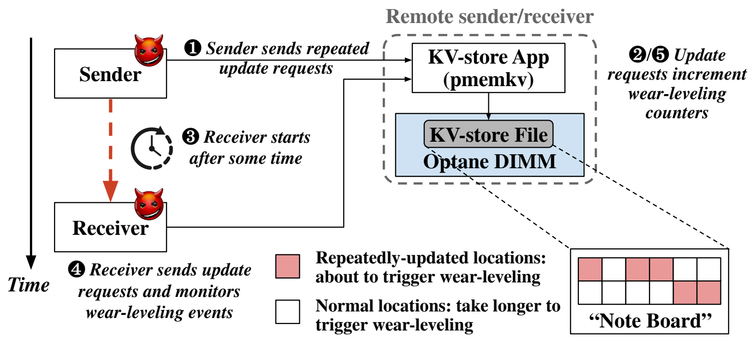

Remote sender/receiver
Sender
Receiver
Time
❶ Sender sends repeated update requests
KV-store App (pmemkv)
KV-store File
Optane DIMM
❷/❺ Update requests increment wear-leveling counters
❸ Receiver starts after some time
❹ Receiver sends update requests and monitors wear-leveling events
Repeatedly-updated locations: about to trigger wear-leveling
Normal locations: take longer to trigger wear-leveling
"Note Board"

Figure 5.17: Remote Note Board attack.

and recent remote covert channels, based on memory deduplication [277], and file-system deduplication [278], respectively. Compared to the DDIO covert channel on RDMA-capable network interface [110], our bandwidth is lower. However, as we show in the next section, compared to their work we achieve a significantly higher accuracy in the side-channel scenario both works evaluate. Furthermore, as the timing difference is strong, there is no additional amplification of the signal required [279–282]. There have also been other remote timing attacks, however, they did not report the capacity of their covert channel [283–288].

5.6 Remote Note Board Attack

In this section, we describe another case study of a remote covert channel based on wear-leveling. As Optane remains data over time and across power cycles, it is likely that the wear-leveling metadata (e.g., counters) is also persistent. Thus, we evaluate whether the sender can leave a message on Optane and let the receiver read from it later or after reboot.

5.6.1 Attack Model

Figure 5.17 presents the attack model, where we assume a system setup similar to the attack model in Section 5.5.1, where the sender and the receiver are located in different servers and connected to a common KV-store server via the network. We consider a web-API, where both the sender and the receiver can update certain values (i.e., common keys in the KV-store) but cannot read the full message. One example can be a survey with each field stored in an Optane-backed KV-store, where users can repeatedly update via resubmission. However, unlike the real-time covert channel, the sender in this scenario first leaves the message and the receiver reads the message later to stay more stealthy during transmission.

5.6.2 Attack Design

The attack takes in the following procedure. Initially, the sender issues repeated updates to the `pmemkv` KV-store server (step ❶). By controlling the number of updates to the same key, the sender can set the wear-leveling counter to a certain level (step ❷). Moreover, the sender repeats this procedure for different keys to encode the whole message. Then, the sender goes offline, and the receiver attempts to recover the message after some waiting time (step ❸). The receiver obtains the message by probing different keys (step ❹) and counting the number of updates needed to trigger a wear-leveling event (step ❺). A small number of update requests indicate the sender has left a bit value of 1 by issuing a large number of initial update requests (otherwise, the bit is 0). As the sender leaves a message on Optane and the receiver retrieves it after some time, we call it the *Note Board* attack.

Challenges. First, the Note Board attack also faces the same challenges as our remote covert channel (Section 5.5), regarding the request intensity and update granularity. We handle them in the same way as Section 5.5.2. However, the Note Board attack transmits message through a range of locations, not a single block. Thus, a new challenge is that a wear-leveling remapping may reset other adjacent ones, as the remote sender has no control over memory allocation on the server.

Solutions. To overcome the new challenge, the sender allocates a large value of 4 kB. During the update, the sender sets a 512 B value within the 4 kB block. This way, the distance between two updates is at least 4 kB, reducing the chance of remapping interference. However, the persistent memory allocator contains metadata and may pad allocated blocks—the actual size of a value can exceed 4 kB and still cause interference. Therefore, the sender further uses multiple key-values to encode one bit in the message as redundancy.

5.6.3 Attack Setup

We use the same system as Section 5.5.3, with the KV-store server (based on Intel's `pmemkv` [235]) running on a prefetcher-enabled CPU. We also evaluate the channel with *no*, *low*, *medium*, and *high* noise levels (methodology in Section 5.4.3). For each noise level, we test three time gaps: 1 minute, 1 hour, and 1 day. In addition, we include a *reboot* scenario to evaluate this attack across power cycles. Due to the long waiting time, we evaluate 10 tests per setting, where the sender transmits a 100 bit message to the receiver. To store this "Note Board", the sender's keys take 4 MB out of the total 256 MB of the `pmemkv` storage on Optane. We pre-allocate files for all iterations of the attack

Table 5.7: Note Board attack accuracy under different noise types and time gaps ($n = 10$).

Wait Time	Noise	Acc (%)	σ_{Acc}
1 min	No	92.40	2.56 %
1 hour		92.30	3.61 %
1 day		92.77	3.35 %
1 min	Low	92.70	2.74 %
1 hour		92.00	4.00 %
1 day		91.70	3.17 %
1 min	Med	90.90	6.50 %
1 hour		90.90	3.82 %
1 day		90.10	2.48 %
1 min	High	89.70	3.02 %
1 hour		89.30	3.88 %
1 day		89.60	3.72 %
1 min	High (100 % Update)	89.40	4.25 %
1 hour		86.70	3.26 %
1 day		82.90	7.38 %
—	Reboot	91.20	3.46 %

to prevent interference from prior runs through the wear-leveling counters, as the Optane locations used by one iteration may be allocated to the next one.

5.6.4 Results

Table 5.7 presents our accuracy results. In scenarios without noise, the message can be successfully retrieved from the wear-leveling-based Note Board at a high accuracy of more than 92 %, even after 1 day of wait time. Although the background noise (i.e., other KV-store activities) has a strong interference in the keystroke attack (Section 5.4), this wear-leveling-based method does not degrade much due to the noise. Even under a high noise level, the accuracy can still be as good as 89 %, and is insensitive against waiting time. There are two main reasons: (1) the wear-leveling latency is two orders of magnitude higher than normal access latencies, which is hard to be interfered during retrieval, and (2) it takes a large number of updates ($>$ 10 000, Section 5.2.5) under normal write access patterns—normal background activities rarely cause remap of Optane pages within the Note Board region even after 1 day. We further increase the write intensity of the background activity, from 20 % update requests (Section 5.4.3) to 100 %. Although the accuracy gets noticeably lower, E.g., 82.90 % after 1 day of wait time, it still remains usable. Moreover, the Note Board remains accurate after reboot (91.20 % accuracy), which confirms that the wear-leveling metadata is persistent. We conclude that the Note Board attack is robust against normal interference, which makes it harder to defend against. We propose a defense mechanism in Section 5.7.2.

Comparison with existing attacks. While there have been many remote covert channels already, as we have discussed in Section 5.5.4, they differ from our Note Board attack as they are usually not persistent and asynchronous. Instead, they are temporal, and require the sender and the receiver to collude and transmit data synchronously.

5.7 Discussion

In this section, we discuss future works and our proposal for defense mechanisms.

5.7.1 Future Works

Other Optane-based side-channel attacks. In this work, we have provided the basic attack primitives and presented four case studies. As Optane becomes more widely used, we expect more use cases. For example, Optane can serve as the storage backend for general applications [31, 35] and a large memory for scientific computing [289, 290]. We expect future research to explore other types of side-channel attacks, such as workload detection, based on our attack primitives.

Attack on different Optane configurations. In this work, we study one Optane DIMM installed alongside the DRAM. Optane also allows multiple DIMMs to be interleaved and work as a single, large device [291], similar to RAID-0 of hard drives. As writes are divided among different DIMMs, we expect different internal caching behaviors. Besides the persistent use cases, Optane memory can also serve as a large volatile memory (i.e., Intel's Memory Mode [291]). We expect future research to investigate these alternative configurations.

5.7.2 Defense Mechanisms

In this section, we briefly describe three proposals for mitigating side-channel attacks on Optane memory.

Mitigation of side channels from internal buffers. The internal buffers, RMW and AIT, are structures that can lead to side channels. Similar to the defense mechanism for CPU cache side channels, it is possible to divide these buffers for each application/user and provide isolation [292–295]. Likewise, better replacement policies and hashing schemes [296–299] may also mitigate the side channels of buffers in the Optane memory.

Attack primitive detection. Similar to attacks on CPU caches, attacks on Optane memory also follow certain patterns, such as Prime+Probe. Therefore, prior solutions for detecting cache attacks can also be useful for Optane memory [300–303]. Upon detection of repeated access patterns, the hardware can throttle the accesses speed to Optane or change the replacement policy of internal buffers (e.g., force buffer flush), in order to break the side channel. However, the wear-leveling channel, which can be exploited using normal key-value updates, is hard to detect. Next, we describe a proposal for mitigating the wear-leveling channel.

Wear-leveling timing mitigation. Wear-leveling causes a significant access delay, likely because accesses cannot continue when an Optane-internal page is being remapped. Therefore, one mitigation is to eliminate the stop-the-world wear-leveling. Instead, we propose an adaptive wear-leveling mechanism. First, the device can perform wear-leveling early when the page is not being accessed but thresholds are about to be reached, effectively working as a "garbage collector" in the background. Second, as Optane (or other persistent memory [31, 304, 305]) tends to have a high write endurance level (e.g., 10^7 per cell [306]), *wear-leveling is not an urgent event*; Optane can also postpone wear-leveling when there are continuous writes to the same page. When the series of writes complete, wear-leveling can happen in the background without hurting the performance or leaking sensitive information. Note that, by keeping track of the wear-leveling counters, the Optane controller can still balance the write endurance of different memory pages. For example, pages with more accumulative writes will have a lower wear-leveling threshold. This way, it will be substantially harder for the attacker leave a message on Optane using the wear-leveling counters.

Chapter 6

Related Works

In this chapter, we present a survey of related works on software and hardware systems for persistent memory.

6.1 Software Systems for Persistent Memory

6.1.1 Persistent Memory Libraries

As Chapter 3 has shown, programming for PM is challenging. To reduce the programming burden, developer from the industry and academia have developed PM libraries, such as PMDK [35], Mnemosyne [36], and NV-Heaps [37] that provide transactional interface to ensure failure-atomicity of updates to PM. To maintain crash consistency, these transactional approaches typically use undo and/or redo logging to maintain a consistent copy of the persistent data that is under in-place updates. Instead of performing in-place updates, out-of-place updates can also ensure crash consistency. For example, MOD [38] performs atomic updates for failure-recovery. These PM libraries make programming for PM systems easier but still require a correct crash consistency algorithm and a good understanding of the library support. Our testing works in Chapter 3 can be adapted to these PM libraries and assist programmers to implement correct programs. Besides library support, there are also frameworks that convert legacy code to a persistent version. For example, Atlas [159], NVthreads [172], PMThreads [307], and SFR [166] use the synchronization primitives in multithreaded programs as hints to covert existing programs into a crash-consistent, PM-based version. iDO [168] and Clobber-NVM [308] take a different approach. They identify idempotent code regions and convert them into failure-atomic code regions for PM. Although these tools perform automated code conversion, PM-specific testing tools can still help to ensure end-to-end correctness, as these frameworks and their converted code boil down to low-level PM accesses.

6.1.2 File systems for Persistent Memory

Conventional storage systems usually use SSDs and HDDs as the storage medium. The introduction of PM technologies provide an opportunity to perform load/store instructions directly to the persistent data. However, software systems need to be redesigned to leverage this direct access capability. To utilize the existing file system interface to access PM, there have been works that develop PM-optimized file systems. For example, PMFS [130] is a PM-optimized POSIX file system from Intel that optimizes for PM's byte-addressable accesses. NOVA [9] is a log-based file system optimized for PM-DRAM hybrid memory systems. SplitFS [11] reduces the system call overhead by splitting responsibilities of data management from metadata management, where access to data are performed directly in user-space while metadata is operated in the kernel. There are also many other file systems for PM, such as BPFS [309], Mojim [310], Strata [311], NOVA-Fortis [10], and SCMFS [312] are file systems. Regardless of the interface, these file systems internally still rely on direct management of persistent data. Thus, testing frameworks can assist the development of these PM-optimized file systems.

6.1.3 Testing for Persistent Memory Software

There have been specialized testing tools to help programmers detect crash consistency and performance bugs in PM programs. For example, Intel has developed Pmemcheck [128] and Persistence Inspector [129] on top of dynamic instrumentation tools to trace PM operations and perform testing. These tools are specific to Intel's PMDK library and have limited testing scope. To improve the scope and library support, we develop PMTest [25] that supports a wider range of PM software systems and cover more buggy scenarios. Further, we develop XFDetector [26] that extends the testing scope by reasoning about the program execution before and after the failure. To support these testing tools, we also develop PMFuzz [27] that uses the fuzzing approach to efficiently generate PM-specific test cases. Follow to our works, there have been other testing tools designed for PM systems. For example, instead of runtime testing, Agamotto [313] performs symbolic execution, and Witcher [314] uses a combination of static and dynamic trace analysis. Jaaru [315] is a testing tool that uses model checking. Compared to Intel's modeling checking tool, Yat [184], Jaaru significantly improves the execution time. These tools, including the testing works in this thesis, primarily focus on a single-thread. There have also been recent works that target crash consistency bugs in multithreaded programs, such as PMRace [316] and DURINN [317].

6.2 Hardware Systems for Persistent Memory

6.2.1 Memory Persistency

Memory persistency models ensure the order in which writes become persistent. Pelly et al. first propose memory persistency [39] and followup research continue to optimize the performance of persistency models. For example, DPO [40] is a more relaxed persistency model that decouples the order of writes to PM from the volatile execution; HOPS [1] separates fences for ordering and durability to reduce the blocking overhead due to persistence; PMEM-Spec [41] speculatively breaks the ordering constraints of PM writes and recovers from misspeculation as if it is a system failure; Themis [42] extends the x86 persistency model to provide ordering guarantees for common undo logging programming pattern, without needing fences. These persistency models are provided by the hardware system and are critical for the upper-level software to ensure the crash consistency guarantees. Our notion of counter-atomicity for encrypted PM systems [28] (Chapter 4A) is orthogonal to persistency models. Instead of enforcing the ordering of writes to PM, counter-atomicity ensures that the data and counter of the same write access are persisted atomically. Therefore, counter-atomicity is applicable to any persistency model.

6.2.2 Hardware-based Crash Consistency Mechanisms

Providing the crash consistency guarantee is another important aspect in PM systems. Prior works have proposed and implemented a variety of software and hardware solutions to maintain crash consistency. Hardware-based mechanisms include implementations of low-level primitives such as DPO [40] and HOPS [1], and high-level hardware transactions such as Kiln [45], ThyNVM [47], JUSTDO Logging [156] and ATOM [46]. Software-based solutions, such as NV-Heaps [37], Mnemosyne [36], REWIND [158], Intel's PMDK [35], LSNVMM [206], etc., abstract away the low-level crash consistency mechanism and provide a high-level software interface for programmers to manage their persistent data. There are also PM-optimized file systems, such as Intel's PMFS [8], BPFS [161], NOVA [9], and SCMFS [195]. Our software-hardware co-design, Janus [30] (Chapter 4B), can be integrated with these crash consistency mechanisms to improve performance.

For example, PM transactions can overlap the latency of backend memory operations with other transactional steps using our pre-execution technique.

6.2.3 Security Guarantees in PM Hardware Systems

There have been existing proposals that provides security guarantees. For example, there have been works targeting PM encryption [28, 50, 56, 77, 199, 318], designs that target data integrity [30, 57, 59, 319–322], and hardware support for the recoverability of PM systems with security guarantees [28, 318, 323–325]. These works focus on integrity and confidentiality of data on PM. However, these PM hardware system are still susceptible to side-channel attacks. In this thesis, we take a step further to exploit the side-channel vulnerabilities in the existing Optane PM system (Chapter 5).

Chapter 7

Conclusions

Persistent memory (PM) technologies, such as Intel's Optane PM, provide a class of high-performance, byte-addressable, and durable memory. These new features allow the software to directly manage their persistent data in memory, without going through the file system indirections. We identify that integrating this new class of memory requires a redesign across the system stack. First, it is hard and error-prone to implement crash-consistent programs that directly manage persistent data. Second, PM requires the integration of different memory and storage supports, such as memory encryption and integrity verification that secure the data and memory compression that improves the bandwidth. However, a naive integration can lead to performance degradation and even break the crash consistency guarantees. Third, there are also other vulnerabilities in real PM systems. For example, the hardware structures in the commercially-available Optane PM can be leveraged by side-channel attacks.

My thesis provides system supports to overcome these new challenges. We hypothesize that a whole-system-level redesign, from programming support to hardware, that ensures correctness, security, and high performance, is necessary in order to integrate persistent memory into practical systems.

To overcome the difficulties in ensuring the correct implementation of the crash consistency guarantee, my research develops a series of testing tools to detect bugs that can lead to inconsistencies after failure. More specifically, PMTest is a runtime testing tool that traces program execution and checks whether the execution violates the requirements for persistence. Beyond runtime testing, XFDetector further extends the scope to the end-to-end execution—before and after the failure. Moreover, we develop a test case generator, PMFuzz, that efficiently generates test cases for PM programs. Overall, these testing works have detected 18 bugs in PM-based programs that are developed by the industry.

On the hardware side, my research targets both the efficiency and security aspects of PM. First, we propose a scheme of metadata atomicity to make sure that the metadata associated with PM operations becomes persistent atomically with the data. This way, even in case of a failure, the recovery procedure can still use the correct metadata to restore the data. For example, in counter-mode encryption, making the counter atomically persistent with the encrypted data guarantees the data read during recovery can always be decrypted with the correct counter. Besides the consideration of crash consistency, performance is another key aspect, especially when there is a combination of operations associated with PM, such as encryption, integrity verification, and data deduplication. We provide a hardware design, Janus, that parallelizes these operations and pre-executes them as soon as the address and data of a future persist operation are known.

Finally, my thesis also exploits security vulnerabilities in the existing Optane PM. We find out that the in-Optane buffering, caching, and wear-leveling mechanisms can be utilized as side channels that leak secret information about the program. For example, an attacker can monitor the Optane access latency to reveal the keystroke input to an Optane-backed key-value store; a sender can covertly leave a message on an Optane-backed key-value store for a day through the timing variation from the wear-leveling mechanism. Our research has shown that Optane PM has unique side-channel vulnerabilities that need to be mitigated in future generations.

Moving forward, there are many new areas in PM research. One direction is to adapt PM into larger-scale systems. As PM has a low memory access latency and high capacity, it can accelerate storage-class applications. However, in large-scale systems, accesses to storage are usually from remote servers, placing the network latency on the critical path and overshadowing the low PM access latency. Thus, a research direction is to redesign the large-scale storage system to better leverage the low access latency of PM. Besides serving as a persistent storage device, PM can also provide large memory capacity (e.g., tens of TBs in a server) at a lower cost. Therefore, another research direction is to leverage the capacity advantages of PM to enable workloads that have large-memory footprints. The higher PM capacity can enable applications that would not be possible to perform on a single server, without using memory swap or remote memory. Finally, as my research has shown that it is possible to perform side-channel attacks through the Optane PM, a future direction is to mitigate these vulnerabilities, such as timing side channels over the buffers and the wear-leveling mechanism.

Bibliography

[1] 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 Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[2] 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.

[3] Intel. Intel 64 and IA-32 architectures software developer's manual. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, 2019.

[4] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

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

[6] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *Proceedings The 21st IEEE Symposium on High Performance Computer Architecture*, 2015.

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

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

[9] 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.

[10] 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.

[11] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

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

[13] Intel. `https://software.intel.com/content/www/us/en/develop/articles/code-sample-enable-your-application-for-persistent-memory-with-mysql-storage-engine.html`.

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

[15] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. NVMcached: An NVM-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (ApSys)*, 2016.

[16] Katelin A. Bailey, Peter Hornyack, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)*, 2013.

[17] Whitfield Diffie and Martin Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, 1979.

[18] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and Modeling Non-Volatile Memory Systems. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[19] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.*, 2020.

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[21] Jialiang Zhang, Nicholas Beckwith, and Jing Jane Li. GORDON: Benchmarking Optane DC Persistent Memory Modules on FPGAs. In *IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021.

[22] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its on-DIMM Buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys)*, 2022.

[23] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security & Privacy (S&P)*, 2019.

[24] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[25] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the*

Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.

[26] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[27] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. PMFuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

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

[29] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors OS- and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[30] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. Janus: Optimizing memory and storage support for non-volatile memory systems. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[31] 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.

[32] Cong Xu, Dimin Niu, Naveen Muralimanohar, Norman P. Jouppi, and Yuan Xie. Understanding the trade-offs in multi-level cell ReRAM memory design. In *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013.

[33] Usharani Upadhyayula. Quick start guide: Provision intel optane dc persistent memory. `https://software.intel.com/content/www/us/en/develop/articles/quick-start-guide-configure-intel-optane-dc-persistent-memory-on-linux.html`, 2019.

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

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

[36] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memeory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[37] 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 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[38] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally ordered durable datastructures for persistent memory. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

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

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

[41] Jungi Jeong and Changhee Jung. PMEM-Spec: Persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[42] S. M. Shahri, S. Armin Vakil Ghahani, and A. Kolli. (Almost) Fence-less persist ordering. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[43] David Mulnix. Intel Xeon Processor D product family technical overview. `https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview`.

[44] Intel. eADR: New opportunities for persistent memory applications. `https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html`, 2021.

[45] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[46] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *Proceedings of The 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[47] 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 Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[48] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[49] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *MICRO*, 2003.

[50] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. DEUCE: Write-efficient encryption for non-volatile memories. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[51] Weidong Shi, Hsien-Hsin S. Lee, Mrinmo Ghosh, Chenghuai Lu, and Alexandra Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[52] Chenyu Yan, Brian Rogers, Daniel Englender, Yan Solihin, and Milos Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[53] Brain Rogers, Yan Solihin, and Milos Prvulovic. Memory predecryption: Hiding the latency overhead of memory encryption. In *Workshop on Architectural Support for Security and Anti-Virus*, 2004.

[54] Moinuddin K. Qureshi, Michele Franchescini, Vijayalakshmi Srinivasan, Luis Lastras, Bulent Abali, and John Karidis. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[55] Siddhartha Chhabra and Yan Solihin. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[56] Pengfei Zuo and Yu Hua. SecPM: A secure and persistent memory system for non-volatile memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[57] M. Ye, C. Hughes, and A. Awad. Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[58] Joydeep Rakshit and Kartik Mohanram. ASSURE: Authentication scheme for secure energy efficient non-volatile memories. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC)*, 2017.

[59] S. Swami and K. Mohanram. ARSENAL: Architecture for secure non-volatile memories. *IEEE Computer Architecture Letters*, 17(2):192–196, July 2018.

[60] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[61] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, 2003.

[62] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *The Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[63] D. Williams and Emin Gun Sirer. Optimal parameter selection for efficient memory integrity verification using merkle hash trees. In *Third IEEE International Symposium on Network Computing and Applications, 2004. (NCA 2004). Proceedings.*, 2004.

[64] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[65] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *Pro-*

ceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security (CCS), 2013.

[66] L. Ren, C. W. Fletcher, X. Yu, M. van Dijk, and S. Devadas. Integrity verification for path oblivious-RAM. In *IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

[67] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[68] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [Nearly] free recursion and integrity verification for position-based oblivious RAM. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[69] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[70] R. Wang, Y. Zhang, and J. Yang. Cooperative Path-ORAM for effective memory bandwidth sharing in server settings. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[71] R. Wang, Y. Zhang, and J. Yang. D-ORAM: Path-ORAM delegation for low execution interference on cloud servers with untrusted memory. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[72] A. Shafiee, R. Balasubramonian, M. Tiwari, and F. Li. Secure DIMM: Moving ORAM primitives closer to memory. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[73] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2010.

[74] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[75] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[76] C. Wang, Q. Wei, J. Yang, C. Chen, Y. Yang, and M. Xue. NV-Dedup: High-performance inline deduplication for non-volatile memory. *IEEE Transactions on Computers (TC)*, 67(5):658–671, 2018.

[77] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[78] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith. Memory expansion technology (MXT): Software support and performance. *IBM Journal of Research and Development*, 45(2):287–301, March 2001.

[79] Alaa Alameldeen and David Wood. Frequent pattern compression: A significance-based compression scheme for L2 caches. Technical Report 1500, Computer Sciences Dept., UW-Madison, 2004.

[80] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[81] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010.

[82] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. C-Pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1196–1208, Aug 2010.

[83] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[84] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. HICAMP: Architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[85] G. Pekhimnko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[86] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis. MemZip: Exploring unconventional benefits from memory compression. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[87] Dhananjoy Das, Dulcardo Arteaga, Nisha Talagala, Torben Mathiasen, and Jan Lindström. NVM compression—hybrid flash-aware application level compression. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[88] Stuart Schechter, Gabriel H. Loh, Karin Strauss, and Doug Burger. Use ECP, not ECC, for hard failures in resistive memories. In *Proceeding of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010.

[89] Y. Cassuto, M. Schwartz, V. Bohossian, and J. Bruck. Codes for asymmetric limited-magnitude errors with application to multilevel flash memories. *IEEE Transactions on Information Theory*, 56(4), 2010.

[90] W. Wen, Y. Zhang, Mengjie Mao, and Y. Chen. State-restrict MLC STT-RAM designs for high-reliable high-performance memory system. In *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014.

[91] R. Naseer and J. Draper. Parallel double error correcting code design to mitigate multi-bit upsets in SRAMs. In *ESSCIRC 2008 - 34th European Solid-State Circuits Conference*, 2008.

[92] Qingan Li, Yanxiang He, Yong Chen, Chun Jason Xue, Nan Jiang, and Chao Xu. A wear-leveling-aware dynamic stack for PCM memory in embedded systems. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2014.

[93] Huizhang Luo, Qingfeng Zhuge, Liang Shi, Jian Li, and Edwin H.-M. Sha. Accurate age counter for wear leveling on non-volatile based main memory. *Design Automation for Embedded Systems*, 2013.

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

[95] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[96] Jianhui Yue and Yifeng Zhu. Accelerating write by exploiting PCM asymmetries. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[97] Kosuke Suzuki and Steven Swanson. The non-volatile memory technology database (NVMDB). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015. http://nvmdb.ucsd.edu.

[98] Seokin Hong, Prashant Nair, Bulent Abali, Alper Buyuktosunoglu, Kyu Hyoun Kim, and Michael Healy. Attache: Towards ideal memory compression by mitigating metadata bandwidth overheads. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[99] Esha Choukse, Mattan Erez, and Alaa R. Alameldeen. Compresso: Pragmatic main memory compression. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[100] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *Annual International Cryptology Conference (CRYPTO)*, 1996.

[101] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security & Privacy (S&P)*, 2015.

[102] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[103] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[104] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.

[105] Berk Gülmezoğlu, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. A Faster and More Realistic Flush+Reload Attack on AES. In *6th International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2015.

[106] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[107] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. *Proceedings on Privacy Enhancing Technologies (PETS)*, 2015.

[108] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 Strikes Back. In *Proceedings of the 10th ACM symposium on Information, computer and communications security (AsiaCCS)*, 2015.

[109] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[110] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *IEEE Symposium on Security & Privacy (S&P)*, 2020.

[111] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019. Extended classification tree and PoCs at https://transient.fail/.

[112] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[113] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[114] Stephan van Schaik, Alyssa Milburn, Sebastian österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE Symposium on Security & Privacy (S&P)*, 2019.

[115] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[116] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Symposium on Security & Privacy (S&P)*, 2020.

[117] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *European Symposium on Research in Computer Security (ESORICS)*, 2019.

[118] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[119] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[120] Fumiyasu Ishibashi. Introducing Optane DC persistent memory. `http://www.ipsj.or.jp/sig/os/index.php?plugin=attach&refer=ComSys2019&openfile=ComSys2019-IntelDCPMMver1.0.pdf`, 2019.

[121] Li-Pin Chang and Chun-Da Du. Design and Implementation of an Efficient Wear-Leveling Algorithm for Solid-State-Disk Microcontrollers. *ACM Trans. Des. Autom. Electron. Syst.*, 2010.

[122] Michael Wu and Willy Zwaenepoel. ENVy: A non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[123] Dawoon Jung, Yoon-Hee Chae, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A Group-Based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems (CASES)*, 2007.

[124] Intel. Product brief: Data center Intel Optane DC persistent memory. `https://www.mouser.cn/datasheet/2/612/optane-dc-persistent-memory-brief-1710301.pdf`, 2019.

[125] 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.

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

[127] 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.

[128] Eduardo Carellan. Discover persistent memory programming errors with pmemcheck. `https://software.intel.com/content/www/us/en/develop/articles/discover-persistent-memory-programming-errors-with-pmemcheck.html`, 2018.

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

[130] 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.

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

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

[133] Intel. Btree: snapshot node before modifying it (PMDK). `https://bit.ly/2BLZHCo`, 2018.

[134] Intel. Btree: remove not needed snapshot (PMDK). `https://bit.ly/367Jc1m`, 2018.

[135] 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 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[136] 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.

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

[138] 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.

[139] 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.

[140] 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), October 2014.

[141] 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.

[142] 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.

[143] 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.

[144] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

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

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

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

[148] 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.

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

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

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

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

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

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

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

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

[157] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

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

[159] 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.

[160] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing checkpoints using NVM as virtual memory. In *IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.

[161] 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 Symposium on Operating Systems Principles (SOSP)*, 2009.

[162] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[163] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[164] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent Computing: Challenges and Opportunities. In *2nd Summit on Advances in Programming Languages (SNAPL)*, 2017.

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

[166] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[167] David E. Lowell and Peter M. Chen. Free transactions with rio vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[168] Qingrui Liu, Joseph Lzraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[169] E. R. Giles, K. Doshi, and P. Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[170] Michael Wu and Willy Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[171] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous checkpointing of htm transactions in nvm. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2017.

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

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

[174] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *IEEE 30th International Conference on Data Engineering (ICDE)*, 2014.

[175] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

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

[177] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

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

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

[180] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[181] David Drysdale. Coverage-guided kernel fuzzing with syzkaller, 2016.

[182] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

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

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

[185] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834, 2019.

[186] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[187] Intel. PMDK man page: libpmem. `https://pmem.io/pmdk/manpages/linux/master/libpmemobj/libpmemobj.7.html`.

[188] Michael Boelen. Linux and ASLR: kernel/randomize_va_space. `https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/`, 2016.

[189] Yan Shoshitaishvili. Preeny. `https://github.com/zardus/preeny/`.

[190] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[191] Intel. Pmdk mapcli. `https://github.com/pmem/pmdk/blob/master/src/examples/libpmemobj/map/mapcli.c`.

[192] AFLplusplus. American fuzzy lop plus plus (afl++). `https://aflplus.plus/`.

[193] Laf Intel. Circumventing fuzzing roadblocks with compiler transformations. `https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/`, 2016.

[194] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[195] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[196] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The linux kernel library. In *9th RoEduNet IEEE International Conference*, 2010.

[197] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[198] William Wang and Stephan Diestelhorst. Persistent atomics for implementing durable lock-free data structures for non-volatile memory (brief announcement). In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 309–311, New York, NY, USA, 2019. Association for Computing Machinery.

[199] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[200] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally. Scaling the power wall: A path to exascale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 830–841, Nov 2014.

[201] ITRS. International technology roadmap for semiconductors: 2005 edition, assembly and packaging. `https://www.semiconductors.org/clientuploads/Research_Technology/ITRS/2005/1_Executive%20Summary.pdf`, 2005.

[202] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: Challenges in and avenues for CMP scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

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

[204] Storage Networking Industry Initiative (SNIA). NVDIMM messaging and FAQ. `https://www.snia.org/sites/default/files/NVDIMM%20Messaging%20and%20FAQ%20Jan%2020143.pdf`.

[205] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

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

[207] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T. Chong. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in NVMs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[208] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[209] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. Hardware supported persistent object address translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[210] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[211] Billy Tallis. The Intel Optane SSD DC P4800X (375GB) review: Testing 3D XPoint performance. `https://www.anandtech.com/show/11209/intel-optane-ssd-dc-p4800x-review-a-deep-dive-into-3d-xpoint-enterprise-performance`, 2017.

[212] Gordon Mah Ung. Optane memory review: Why you may want Intel's futuristic cache in your PC. `https://www.pcworld.com/article/3191706/storage/optane-memory-review-why-you-may-want-intels-futuristic-cache-in-your-pc.html`, 2017.

[213] Intel. Intel Optane SSD DC P4800X series. `https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html`.

[214] Everspin Technologies. Storage solutions achieve greater performance with MRAM. `https://www.everspin.com`.

[215] Arthur Sainio. NVDIMM - Changes are here so what's next? `https://www.snia.org/sites/default/files/SSSI/NVDIMM%20-%20Changes%20are%20Here%20So%20What%27s%20Next%20-%20final.pdf`, 2016.

[216] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`.

[217] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom application transaction processing benchmark. `http://tatpbenchmark.sourceforge.net/`, 2011.

[218] Transaction Processing Performance Council (TPC)). Tpc-c. `http://www.tpc.org/tpcc/default.asp`.

[219] Akashi Satoh and Tadanobu Inoue. ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS. *INTEGRATION, the VLSI journal*, 40(1):3–10, 2007.

[220] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact Rijndael hardware architecture with S-box optimization. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 239–254. Springer, 2001.

[221] Y. Sui, D. Ye, and J. Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, Feb 2014.

[222] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM.

[223] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

[224] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security & Privacy (S&P)*, 2011.

[225] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[226] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[227] Onur Acıiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security (AsiaCCS)*, 2007.

[228] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.

[229] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[230] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. `https://foreshadowattack.eu/foreshadow-NG.pdf`, 2018.

[231] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan & Claypool, 2nd edition, 2013.

[232] Intel. Intel Optane DC Persistent Memory. `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`, 2021.

[233] The Linux Kernel Archives. Direct Access for files. `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`, 2021.

[234] Intel. Persistent Memory FAQ. `https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-faq.html`, 2020.

[235] Intel. pmemkv. `https://github.com/pmem/pmemkv`, 2022.

[236] Intel. Redis. `https://github.com/pmem/redis/tree/3.2-nvml`, 2017.

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

[238] Intel. PMSE - Persistent memory storage engine for MongoDB. `https://github.com/pmem/pmem-rocksdb`, 2018.

[239] Intel. PMEM-RocksDB. `https://github.com/pmem/pmse`, 2018.

[240] Intel. Code Sample: Enable Your Application for Persistent Memory with MySQL Storage Engine. `https://software.intel.com/content/www/us/en/develop/articles/code-sample-enable-your-application-for-persistent-memory-with-mysql-storage-engine.html`, 2019.

[241] Jeff Barr. Now Available – Amazon EC2 High Memory Instances with 6, 9, and 12 TB of Memory, Perfect for SAP HANA. `https://aws.amazon.com/blogs/aws/now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfect-for-sap-hana/`, 2018.

[242] Nan Boden. Available first on Google Cloud: Intel Optane DC Persistent Memory. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory, 2018.

[243] Dave Eggleston. Persistent Memory: Media, Attachment, and Usage. https://www.snia.org/educational-library/persistent-memory-media-attachment-and-usage-2020, 2020.

[244] Andy Rudoff. Persistent Memory Programming Made Easy with pmemkv. https://www.snia.org/sites/default/files/SDC/2019/presentations/PM/Rudoff_Andy_Persistent_Memory_Programming_Made_Easy_with_pmemkv.pdf, 2019.

[245] Steve Scargall. *pmemkv: A Persistent In-Memory Key-Value Store*, pages 141–153. Apress, Berkeley, CA, 2020.

[246] Intel. IPMCTL: Utility for configuring and managing Intel Optane persistent memory modules (PMem). https://github.com/intel/ipmctl, 2021.

[247] Intel. NDCTL: Utility library for managing the libnvdimm. https://github.com/pmem/ndctl, 2021.

[248] Intel. Persistent Memory Programming. https://pmem.io/, 2022.

[249] Frank T. Hady. Faster Access to More Data. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/faster-access-to-more-data-article-brief.html, 2019.

[250] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2015.

[251] Henry Wong. Intel Ivy Bridge Cache Replacement Policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/, 2013.

[252] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In *USENIX Security Symposium*, 2021.

[253] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In *European Symposium on Research in Computer Security (ESORICS)*, 2015.

[254] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning Replacement Policies from Hardware Caches. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[255] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*, 2013.

[256] H. Ghasemzadeh, S. Mazrouee, and M.R. Kakoee. Modified pseudo LRU replacement algorithm. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS)*, 2006.

[257] Tristian “Truth” Brown, Travis Liao, and Jamie Chou. Analyzing the Performance of Intel Optane DC Persistent Memory in App Direct Mode in Lenovo ThinkSystem Servers. `https://lenovopress.com/lp1083.pdf`, 2019.

[258] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[259] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[260] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[261] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. *ACM Transactions on Networking*, 2014.

[262] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *International Conference on Financial Cryptography and Data Security*, 2017.

[263] Benjamin Semal, Konstantinos Markantonakis, Keith Mayes, and Jan Kalbantner. One Covert Channel to Rule Them All: A Practical Approach to Data Exfiltration in the Cloud. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020.

[264] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *IEEE Symposium on Security & Privacy (S&P)*, 2021.

[265] Chen Change Loy. Keystroke100 Dataset. `http://personal.ie.cuhk.edu.hk/~ccloy/downloads_keystroke100.html`, 2021.

[266] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS)*, 2012.

[267] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *ACM SIGMOD Record*, 2011.

[268] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[269] Kehuan Zhang and XiaoFeng Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium*, 2009.

[270] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In *IEEE Symposium on Security & Privacy (S&P)*, 2016.

[271] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security & Privacy (S&P)*, 2012.

[272] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, 2009.

[273] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security Symposium*, 2001.

[274] Laurent Simon, Wenduan Xu, and Ross Anderson. Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. *Proceedings on Privacy Enhancing Technologies*, 2016.

[275] Yohan Muliono, Hanry Ham, and Dion Darmawan. Keystroke Dynamic Classification using Machine Learning for Password Authorization. *Procedia Computer Science*, 2018.

[276] Kwesi Elliot, Jonathan Graham, Yusef Yassin, Trenton Ward, John Caldwell, and Tawab Attie. A Comparison of Machine Learning Algorithms in Keystroke Dynamics. In *International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019.

[277] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote Page Deduplication Attacks. In *Network and Distributed System Security Symposium (NDSS)*, 2022.

[278] Andrei Bacs, Saidgani Musaev, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. DUPEFS: Leaking Data Over the Network With Filesystem Deduplication Side Channels. In *20th USENIX Conference on File and Storage Technologies (FAST)*, 2022.

[279] Rodney Owens and Weichao Wang. Non-interactive OS fingerprinting through memory deduplication technique in virtual machines. In *International Performance Computing and Communications Conference*, 2011.

[280] Mathy Vanhoef and Tom Van Goethem. HEIST: HTTP Encrypted Information can be Stolen through TCP-windows. In *Black Hat US Briefings, Location: Las Vegas, USA*, 2016.

[281] Tom Van Goethem, Mathy Vanhoef, Frank Piessens, and Wouter Joosen. Request and conquer: Exposing cross-origin resource size. In *USENIX Security Symposium*, 2016.

[282] Ben Titzer. What Spectre means for Language Implementers. `https://pliss2019.github.io/ben_titzer_spectre_slides.pdf`, 2019.

[283] Xin-jie Zhao, Tao Wang, and Yuanyuan Zheng. Cache Timing Attacks on Camellia Block Cipher. *Cryptology ePrint Archive, Report 2009/354*, 2009.

[284] Darshana Jayasinghe, Jayani Fernando, Ranil Herath, and Roshan Ragel. Remote cache timing attack on advanced encryption standard and countermeasures. In *Fifth International Conference on Information and Automation for Sustainability (ICIAFs)*, 2010.

[285] Hassan Aly and Mohammed ElGayyar. Attacking aes using bernstein's attack on modern processors. In *International Conference on Cryptology in Africa*, 2013.

[286] Vishal Saraswat, Daniel Feldman, Denis Foo Kune, and Satyajit Das. Remote Cache-timing Attacks Against AES. In *Workshop on Cryptography and Security in Computing Systems*, 2014.

[287] Onur Acıiçmez, Werner Schindler, and Cetin K. Koc. Cache Based Remote Timing Attack on the AES. In *CT-RSA*, 2006.

[288] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security Symposium*, 2020.

[289] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *VLDB Endowment*, 2020.

[290] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *VLDB Endowment*, 2020.

[291] Intel. Intel Optane DC persistent memor – Quick start guide. `https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf`, 2020.

[292] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, 2007.

[293] Dan Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *Cryptology ePrint Archive, Report 2005/280*, 2005.

[294] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[295] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security Symposium*, 2019.

[296] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[297] Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[298] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), 2011.

[299] Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and Efficient Elimination of Cache and Timing Side Channels. *arXiv:1506.00189*, 2015.

[300] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[301] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A. Adam Ding. SCADET: A Side-Channel Attack Detection Tool for Tracking Prime+Probe. In *IEEE International Conference on Computer-Aided Design (ICCAD)*, 2018.

[302] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. *Cryptology ePrint Archive, Report 2015/1034*, 2015.

[303] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. NIGHTs-WATCH: A cache-based side-channel intrusion detector using hardware performance counters. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2018.

[304] Myoung-Jae Lee, Chang Bum Lee, Dongsoo Lee, Seung Ryul Lee, Man Chang, Ji Hyun Hur, Young-Bae Kim, Chang-Jung Kim, David H Seo, Sunae Seo, U-In Chung, In-Kyeong Yoo, and Kinam Kim. A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta2O(5-x)/TaO(2-x) bilayer structures. *Nature materials*, 2011.

[305] H. Y. Lee, Y. S. Chen, P. S. Chen, P. Y. Gu, Y. Y. Hsu, S. M. Wang, W. H. Liu, C. H. Tsai, S. S. Sheu, P. C. Chiang, W. P. Lin, C. H. Lin, W. S. Chen, F. T. Chen, C. H. Lien, and M.-J. Tsai. Evidence and solution of over-RESET problem for HfOX based resistive memory with sub-ns switching speed and high endurance. In *International Electron Devices Meeting (IEDM)*, 2010.

[306] Kristian Vättö, Ian Cutress, and Ryan Smith. Analyzing Intel-Micron 3D XPoint: The Next Generation Non-Volatile Memory. `https://www.anandtech.com/show/9470/intel-and-micron-announce-3d-xpoint-nonvolatile-memory-technology-1000x-higher-performance-endurance-than-nand`, 2015.

[307] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. PMThreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[308] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-NVM: Log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[309] 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.

[310] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[311] 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.

[312] 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.

[313] Ian Neal, B. Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, S. Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[314] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

[315] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[316] Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. Efficiently detecting concurrency bugs in persistent memory programs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[317] Xinwei Fu, Dongyoon Lee, and Changwoo Min. DURINN: Adversarial memory and thread interleaving for detecting durable linearizability bugs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[318] Fan Yang, Youyou Lu, Youmin Chen, Haiyu Mao, and Jiwu Shu. No Compromises: Secure NVM with crash consistency, write-efficiency and high-performance. In *Proceedings of the 56th Annual Design Automation Conference (DAC)*, 2019.

[319] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. Triad-NVM: Persistency for integrity-protected and encrypted non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[320] Mazen Alwadi, Kazi Abu Zubair, David Mohaisen, and Amro Awad. Phoenix: Towards ultra-low overhead, recoverable, and persistently secure NVM. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[321] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. Persist level parallelism: Streamlining integrity tree updates for secure persistent memory. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[322] Zhengguo Chen, Youtao Zhang, and Nong Xiao. Cachetree: Reducing integrity verification overhead of secure nonvolatile memories. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(7):1340–1353, 2021.

[323] Fan Yang, Youmin Chen, Haiyu Mao, Youyou Lu, and Jiwu Shu. ShieldNVM: An efficient and fast recoverable system for secure non-volatile memory. *ACM Trans. Storage*, 2020.

[324] Pengfei Zuo, Yu Hua, and Yuan Xie. SuperMem: Enabling application-transparent secure persistent memory with low overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[325] K. A. Zubair and A. Awad. Anubis: Ultra-low overhead and recovery time for secure non-volatile memories. In *ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.