Crash Consistency in Encrypted Non-Volatile Main Memory Systems

Sihang Liu[1], Aasheesh Kolli[2,3], Jinglei Ren[4], and Samira Khan[1]

[1]University of Virginia [2]VMware Research [3]Pennsylvania State University [4]Microsoft Research

Abstract

Non-Volatile Main Memory (NVMM) systems provide high performance by directly manipulating persistent data in-memory, but require crash consistency *support to recover data in a consistent state in case of a power failure or system crash. In this work, we focus on the interplay between the crash consistency mechanisms and memory encryption. Memory encryption is necessary for these systems to protect data against the attackers with physical access to the persistent main memory. As decrypting data at every memory read access can significantly degrade the performance, prior works propose to use a memory encryption technique,* counter-mode encryption*, that reduces the decryption overhead by performing a memory read access in parallel with the decryption process using a counter associated with each cache line. Therefore, a pair of data and counter value is needed to correctly decrypt data after a system crash. We demonstrate that counter-mode encryption does not readily extend to crash consistent NVMM systems as the system will fail to recover data in a consistent state if the encrypted data and associated counter are not written back to memory* atomically*, a requirement we refer to as* counter-atomicity.

We show that naïvely enforcing counter-atomicity *for* all *NVMM writes can serialize memory accesses and results in a significant performance degradation. In order to improve the performance, we make an observation that not all writes to NVMM need to be counter-atomic. The crash consistency mechanisms rely on* versioning *to keep one* consistent copy *of data intact while manipulating another version directly in-memory. As the recovery process only relies on the unmodified consistent version, it is not necessary to strictly enforce* counter-atomicity *for the writes that do not affect data recovery. Based on this insight, we propose* selective counter-atomicity *that allows reordering of writes to data and associated counters when the writes to persistent memory* do not *alter the recoverable* consistent state. *We propose efficient software and hardware support to enforce* selective counter-atomicity. *Our evaluation demonstrates that in a 1/2/4/8-core system,* selective counter-atomicity *improves performance by 6/11/22/40% compared to a system that enforces counter-atomicity for all NVMM writes. The performance of our* selective counter-atomicity *design comes within 5% of an ideal NVMM system that provides crash consistency of encrypted data at no cost.*

1. Introduction

The emerging non-volatile memory (NVM) technologies (e.g., PCM, STT-RAM, ReRAM and Intel 3D XPoint [22, 28, 29, 57]) provide disk-like durability with an access latency close to DRAM [12]. These new memory technologies are blurring the difference between storage and memory, making it possible to store and manipulate persistent data in-place *in memory*. Such systems with *non-volatile main memory (NVMM)*, also referred to as *persistent memory* systems improve the performance of various applications by managing persistent data directly in main memory [5, 7, 7, 10, 26, 31, 34, 43, 58, 62, 63, 64].

Two fundamental challenges need to be addressed in a useful NVMM system that provides legacy storage system supports for persistent data. The first one concerns the recoverability of persistent data from memory in a *consistent state* in the event of a system failure (e.g., unexpected power outages, kernel/application crashes). Ensuring the recoverability of data requires a specific ordering of updates to persistent data *all the way to memory*, a guarantee *not provided* by today's systems in order to enable performance optimizations with caching and reordering of accesses in the memory hierarchy [12, 37]. For example, consider adding a new node to a *persistent linked list* in NVMM. The list can become *inconsistent* in the presence of a system failure, if the crash happens *after* the pointer update that adds the new node to the list reaches NVMM, but *before* the node itself is written back to NVMM, a problem referred to as *crash consistency* problem. Recent works use low-level programming interfaces to govern the order of updates to NVMM or high-level transactions (e.g., redo or undo logging mechanisms) to maintain versions of data to ensure crash consistency in NVMM systems [2, 11, 12, 15, 17, 19, 20, 24, 25, 26, 27, 30, 32, 34, 37, 45, 53, 54].

The second, orthogonal challenge in designing NVMM systems concerns with the security of persistent data in memory. Data in any non-volatile device is persistent across system failures by definition and therefore, vulnerable to malicious attackers who have physical access to the devices [8, 38, 61]. Encryption is an effective solution to protect NVM data from the attackers. In an encrypted NVMM 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*, has been adopted for NVMM to reduce the high overhead of decryption latency during a memory read access [3, 38, 44, 47, 60, 61]. 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 [3, 47, 59].

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 NVMM 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 NVMM generates two write requests: one for the encrypted data and the other for the counter. These two writes to NVMM (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 NVMM, 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 NVMM atomically as *counter-atomicity* and argue that the encrypted NVMM systems need to provide support for *counter-atomicity* to ensure crash consistency in NVMM 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 [23, 40, 51]. The *goal* of this work is to design an *NVMM system that enforces counter-atomicity at a low cost and a low overhead*.

We propose a simple hardware mechanism to enforce *counter-atomicity* in an NVMM 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 NVMM 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 NVMM 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 NVM 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 NVMM. 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 NVMM 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 4). We propose necessary software interface and hardware support to *selectively* enforce *counter-atomicity* in an NVMM system (details in Section 5).

To summarize, the contributions of this work are:

- We show that the commonly used counter-mode encryption *does not* extend to the NVMM 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 NVMM 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 NVM 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.

2. Background and Motivation

In this section, we first discuss the crash consistency support for the non-volatile main memory (NVMM) systems and then demonstrate the challenges in providing crash consistency support when NVMM is *encrypted*.

2.1. Crash Consistency for NVMM Systems

Applications running on the *non-volatile main memory* system manipulate persistent data *in-place* in memory with *direct* read and write accesses (e.g., using load/store instructions). Performance optimizations in the memory system, such as caching and writeback mechanisms coalesce and reorder writes to NVMM, and therefore, such a direct manipulation of persistent data *does not* provide any guarantee on the or-

der in which writes reach the persistent memory. This reordering of writes can lead to an *inconsistent state* in the presence of system failures, such as power outages and application/kernel crashes. For example, consider adding a new node to a *persistent* linked list in memory. The insertion operation consists of two kinds of updates: adding the new node and updating the pointer to the new node. The linked list can fail to recover to a consistent state if a power failure happens *after* the pointer to the new node has been persisted in memory, but the write that fills the new node with valid data *did not* persist in NVMM. Ensuring recoverability of persistent data in a *consistent state* after a crash is defined as *crash consistency*. Prior works propose various hardware and software mechanisms to support crash consistency for the NVMM systems [2, 11, 19, 20, 25, 26, 27, 30, 34, 37, 39, 45, 53]. These crash consistency mechanisms can be broadly classified into two categories. The first category of works focuses on providing support for an ordering of memory updates using `persist_barriers`, which guarantee that the preceding writes have persisted prior to any write that comes afterward in the program order [19, 25, 27, 34, 37]. For example, Intel's implementation of the `persist_barrier` writes back specific cache lines (that comes before the barrier) from the cache to the memory controller write queue and guarantees that all accepted writeback requests in the write queue will be persisted to NVMM in the event of a system failure, using a hardware support referred to as asynchronous DRAM refresh (ADR) [19, 21, 33, 41]. The second category of works focuses on providing high-level APIs, such as libraries and transactional interfaces to store persistent data in NVMM to ease the programmers' burden on manipulating persistent objects in memory [2, 11, 20, 24, 26, 30, 39, 45, 53, 65].

2.2. Crash Consistency for Encrypted NVMM Systems

In this work, we demonstrate that directly applying the existing crash consistency mechanism to an encrypted NVMM system *does not* guarantee a consistent recovery of data in case of a power failure or system crash. Encrypting NVMM is highly important for protecting data, as attackers who have physical access to an NVM module can access information stored in the persistent memory [38, 61]. In this section, first, we briefly introduce the encryption techniques for NVMM systems. Second, we discuss the challenge of providing crash consistency in an encrypted NVMM system, and third, we provide a walkthrough example of how data becomes inconsistent while inserting a node to an encrypted persistent linked list.

2.2.1. Encryption Technique. The NVMM systems encrypt/decrypt data on every memory access using an encryption engine located in the memory controller. Unfortunately, memory reads are on the critical path of the program execution and the additional latency to decrypt data after every read miss can significantly degrade the overall performance. In order 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 NVMM systems [14, 44, 47, 60, 61]. 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 1). This OTP is XORed with the plaintext data to generate the encrypted data (shown in Equation 2, Figure 1(a)). As a result, during a memory read access, the OTP is generated using the associated counter while data is still being fetched from NVMM. When the read access completes, the encryption engine XORs this OTP with the fetched encrypted data to generate the plaintext (shown in Equation 3, Figure 1(b)).

$$OTP = En(address|counter, key) \quad (1)$$

$$EncryptedCacheline = OTP \oplus plaintext \quad (2)$$

$$plaintext = OTP \oplus EncryptedCacheline \quad (3)$$

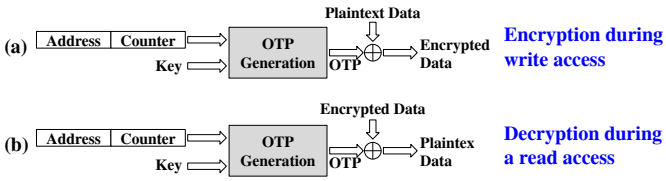


Figure 1: The counter-mode encryption technique: (a) encrypting data during a write access, and (b) decrypting data during a read access.

As counters are required to encrypt and decrypt data for *all* memory accesses, the counters are buffered on-chip in a *counter cache* [59], such that the encryption engine does not need to perform an extra memory read access to fetch the counter value. Figure 2(a) shows the serialized decryption technique that adds additional latency to read accesses and Figure 2(b) shows that the read access is faster with the counter-mode encryption technique as the read access and decryption can be performed in parallel.

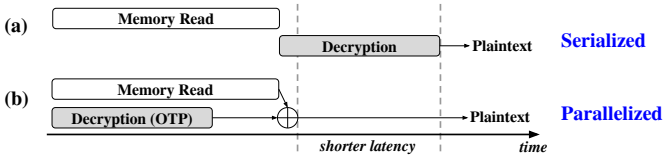


Figure 2: Reduction in latency with the counter-mode encryption technique during a read access.

2.2.2. The Challenge. The main problem with providing crash consistency for an encrypted NVMM system is that each encrypted data is associated with a counter in the counter-mode encryption, but this relationship is *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 NVMM. However, decryption will fail if the versions of data and counter are not in sync (either data or counter in NVMM is stale).

Figure 3 demonstrates that a system failure can result in out-of-sync data and counter. Every write access to NVMM 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 NVMM and before the counter write does, the memory controller would observe a stale counter value upon system recovery, introducing an inconsistency in data recovery, as shown in Figure 3(a). Similar inconsistency occurs if a failure happens after the counter reaches NVMM but the data has not yet been persistent, as shown in Figure 3(b).

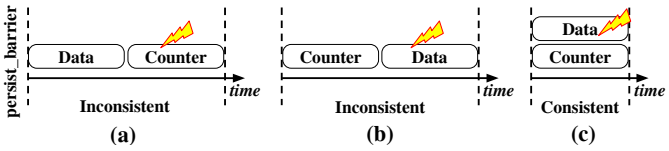


Figure 3: (a) Inconsistent decryption if counter write fails, (b) Inconsistent decryption if data write fails, and (c) Consistent decryption if data and counter writes are atomic.

As $OriginalVal = En(address|counter, key) \oplus EncryptedVal$, then decryption failure happens in these two cases:

$$En(address|counter_{stale}, key) \oplus EncryptedVal_{new} \neq OriginalVal$$
$$En(address|counter_{new}, key) \oplus EncryptedVal_{stale} \neq OriginalVal \quad (4)$$

2.2.3. An Example. Here, we provide an example that shows an *encrypted* persistent linked list can become inconsistent due to out-of-sync data and counter values, if a crash happens while updating the list. Figure 4(a) shows the 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 4(b) 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 NVMM, the values of the `head` pointer and its associated counter in NVMM 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 according to Equation 4 (shown in Figure 4(c)). Potentially, the value of the `head` pointer can become a random number after the incorrect decryption (Equation 3) and the program can mistakenly access a random location in memory. To support crash consistency in an encrypted NVMM, we argue that it is required to enforce an *atomic* behavior of the counter and data writes, which we refer to as *counter-atomicity* (shown in Figure 3(c)).

2.2.4. Our Goal. The goals of this work are: First, demonstrate that the encrypted data and associated counter need to be *atomic* in order to support crash consistency in encrypted NVMM systems (Section 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 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 4).

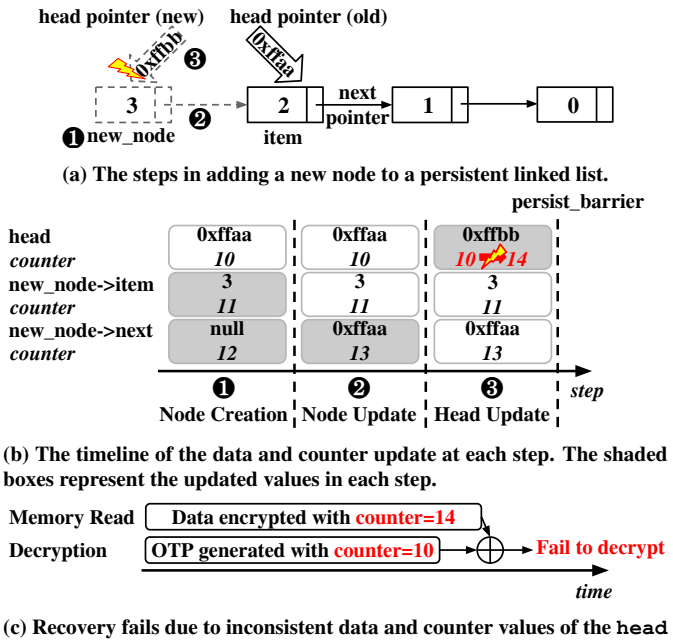


Figure 4: An example of inconsistency while adding a node to a persistent linked list.

3. Counter-Atomicity

The key to maintaining crash consistency in an encrypted NVMM 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 NVMM. 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*.

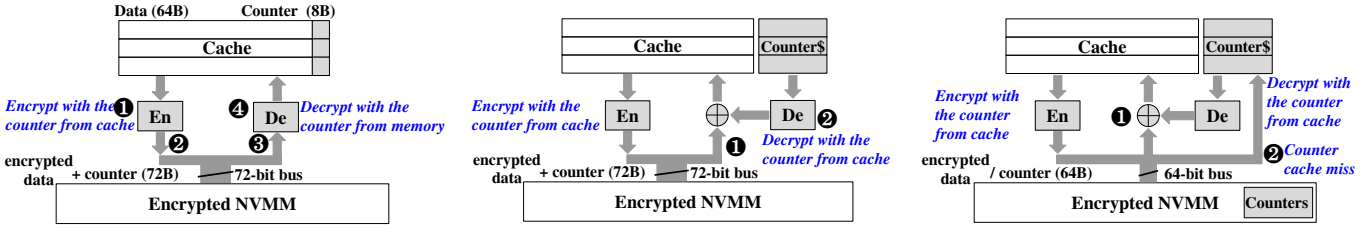
3.1. Requirement

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*.

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.

3.2.1. Challenge 1: How to ensure data and counter reach NVMM at the same time? In today's systems, there is no guarantee that the separate counter and data writes will reach the NVMM 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

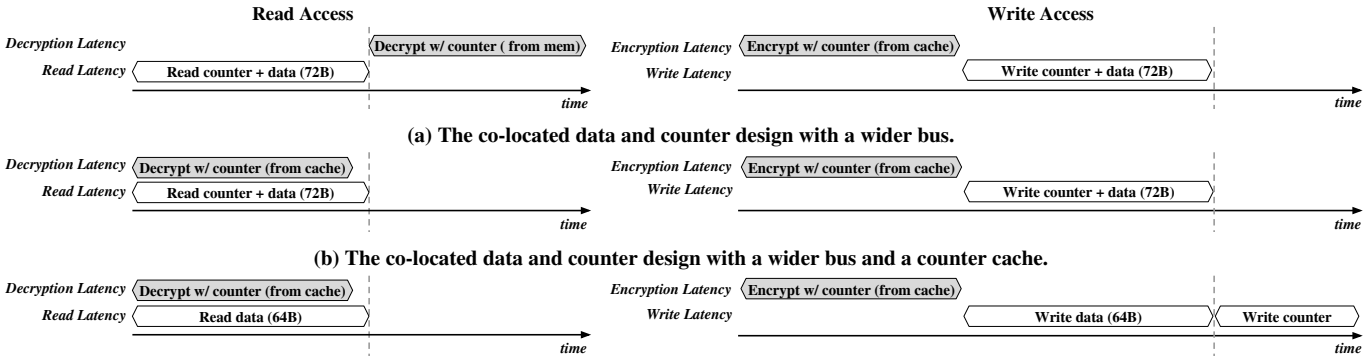


(a) The co-located data and counter design with a wider bus.

(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 5: Different counter-atomic designs.



(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 6: Timeline of read and write accesses with three different design shown in Figure 5.

bus. As every cache line of data needs an 8B counter in the counter-mode encryption (as shown in prior works [3, 44, 59]), 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 5(a) 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 NVMM 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 NVMM (step ❹). Such a serialized design violates the main benefit of the counter-mode encryption technique. Figure 6(a) 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 5(b). The cached counters enable overlapping the decryption process with the read access. While the missed cache line is being fetched from the NVMM (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 6(b) 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 NVMM 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 [23, 40, 51]. We believe that widening the memory bus is not practical and study alternative designs that can enforce *counter-atomicity* with the existing memory interface.

3.2.2. Challenge 2: How to enforce *counter-atomicity* without changing the memory interface? The major drawback in the two aforementioned designs (Figure 5(a) and 5(b)) 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 5.

Figure 5(c) 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 NVMM. 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 6(c) 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 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*.

4. Selective Counter-Atomicity

In this section, first, we discuss the high overhead of enforcing *full counter-atomicity* (Section 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 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 4.3).

4.1. The Overhead of *Full Counter-Atomicity*

Enforcing *counter-atomicity* is necessary to make sure that data in NVM 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 NVMM 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 7(a) 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

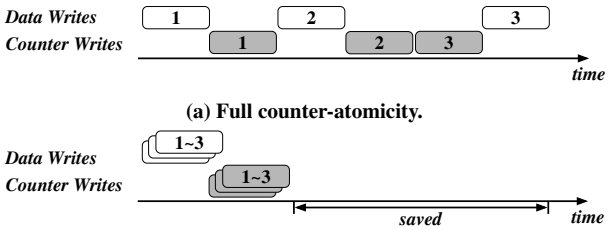


Figure 7: The timeline of write accesses in a *full counter-atomicity* design vs. an *ideal* design that does not enforce *counter-atomicity*.

counter-atomicity. As a result, the write accesses can be reordered, coalesced, and written back in parallel, as shown in Figure 7(b). 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 an NVMM system.

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 [9, 12, 26, 30, 32, 53]. 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 NVMM *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 NVMM *before* they start affecting the recoverability of the system.

Key Insight. Based on the observation that a subset of writes to NVMM *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 NVMM 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 [20, 26, 34]:

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.
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 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 NVMM 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*.

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

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

Figure 8 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 6.3.1). Figure 8(a) 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 8(b)). However, this figure shows that *counter-atomicity* must be enforced for the write accesses in the *commit* stage.

4.3. Definition and Primitives

A *selective counter-atomicity* design has two requirements, *(i)* strictly enforce *counter-atomicity* only for those updates that

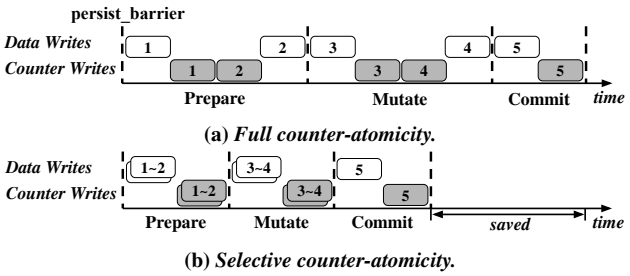


Figure 8: Timeline showing three stages of a transaction with undo-logging under *full counter-atomicity* and *selective counter atomicity*.

immediately affect the recoverability of data in a consistent state, and *(ii)* 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 [19]. 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 4 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 NVMM 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 NVMM *on demand*.

Discussion. The primitives above aims to maximize the performance of the NVMM systems by trading off programmability, similar to the primitives offered by memory persistency models [19, 27, 34]. The responsibility of their correct usage rests with the programmer. However, we expect that expert-crafted libraries, such as NVML [20], will abstract away these low-level primitives from regular programmers.

An example of using the primitives. Figure 9 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 1) are separated by `persist_barrier` to make sure the writes from these stages reach NVMM 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 write-back the latest data and counter values to memory. Sec-

```
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 9: Implementation of an undo-logging transaction with selective counter-atomicity primitives.

ond, 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`.

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 NVMM. Then, we describe the hardware implementation in the memory controller that enforces counter-atomicity.

5.1. System Integration

Figure 10 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 4. The annotation enables the memory controller to differentiate the *counter-atomic* writes from the *non-counter-atomic* ones and write back counter cache lines properly. Next, we discuss the hardware support for counter-atomicity.

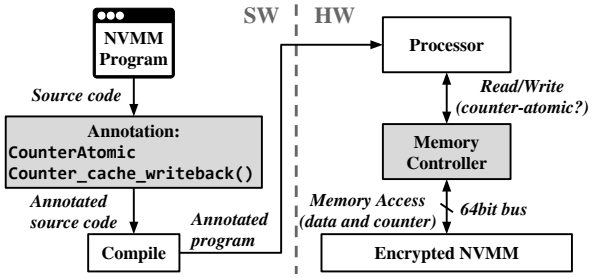


Figure 10: The high-level overview of a system using the *selective counter-atomicity* primitives.

5.2. Hardware Implementation

Figure 11 depicts the memory controller in our design that supports *(i)* encryption and *(ii)* counter-atomicity. The encryption

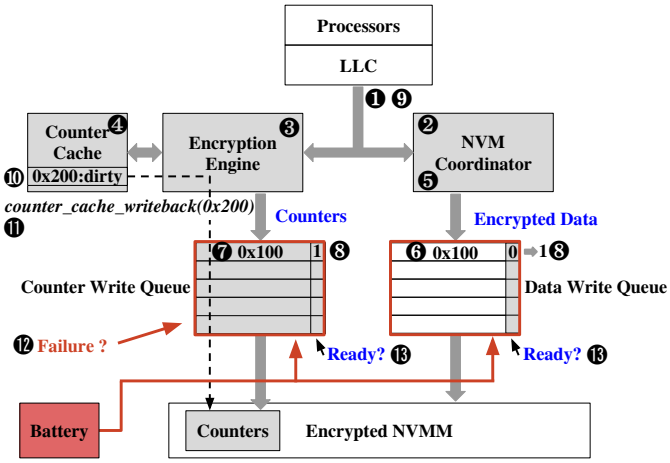


Figure 11: Hardware implementation. The new components are represented with shaded gray, and the persistent structures protected by ADR is shown in red.

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.

5.2.1. 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 *NVM coordinator* performs the read access from NVMM. 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 *NVM 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 NVMM 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 NVMM. 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.

5.2.2. Counter-Atomicity Support. We have shown the hardware support for encrypting and decrypting data. Next, we de-

scribe 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 NVMM with some backup power in case of a power failure [21, 33, 41, 46]. Therefore, this mechanism guarantees that any write request that reaches the write queue will always get persisted to the NVMM. 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 NVMM 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 *NVM 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 6.3.7). 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) [46], 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 11). Step ❶: The processor issues a *counter-atomic* write access to the physical address 0x100. Both the *NVM 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 *NVM 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 *NVM 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 NVMM 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 11). 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.

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.

6.1. Methodology

We model the hardware design described in Section 5 in the cycle-accurate simulator Gem5 [4]. The simulated system consists of x86 out-of-order processors, and an 8GB phase change memory (PCM) [29, 57] with a DDR3 interface (Table 2). Table 2 lists the system parameters used in our evaluation. The *counter cache* in our implementation is 1MB, 16-way set associative. As each counter is 8B, a 1MB counter cache can store 128K counters. However, we show results with different sizes of counter cache in Section 6.3.4. The following are the evaluated designs:

- **No-encryption design.** An NVMM system without any encryption.
- **Ideal design.** An encrypted and crash consistent NVMM system using the counter-mode encryption technique but without any *counter-atomicity* overhead.

Processor	Out-of-Order Cores, 4.0GHz
L1 D/I cache	64KB/32KB per core (private), 8-way
L2 cache	2MB per core (shared), 8-way
Counter cache	1MB per core (shared), 16-way
Memory controller	Data read/write queue: 32/64 entries Counter write queue: 16 entries
Memory	8GB PCM, 533MHz [27], $t_{RCD}/t_{CL}/t_{CWD}/t_{FAW}/t_{WTR}/t_{WR}$ $= 48/15/13/50/7.5/300ns$ [57]
En/decryption	40ns latency [47]

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

- **Co-located data and counter design (Co-located).** An encrypted and crash consistent NVMM 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 3.2.1).
- **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 NVMM using one access co-locating both the data and counter (Section 3.2.1). 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 NVMM 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 3.2.2).
- **Selective counter-atomicity design (SCA).** Similar to the previous design (FCA). However, writes are *counter-atomic* only when necessary (Section 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 NVMM 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.

6.2. Workloads

We evaluate five NVM workloads that manipulate different persistent data structures. Our evaluated workloads are similar to the ones used in prior works on persistent memory systems [11, 25, 27, 39].

- **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.

6.3. Results

We first evaluate the impact of different designs (listed in Section 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.

6.3.1. Single-Core Performance. In this experiment, we compare the performance improvement of different designs. Figure 12 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* 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 NVMM system.

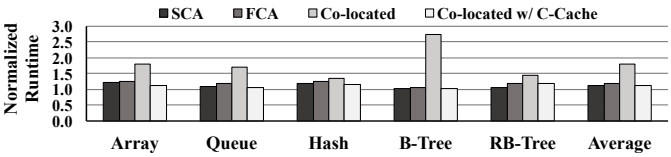


Figure 12: Performance comparison of different design points. The runtime is normalized to the *no-encryption* design (lower is better).

6.3.2. 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 13 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 configura-

tions. 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.

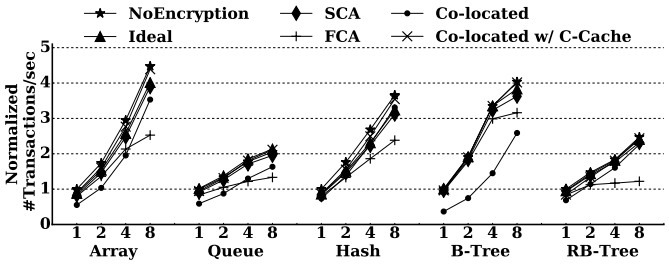


Figure 13: Throughput of multithreaded workloads, normalized to the single-core *no-encryption* design (higher is better).

6.3.3. Write Traffic. Figure 14 shows the write traffic to the NVMM 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 NVMM. 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.

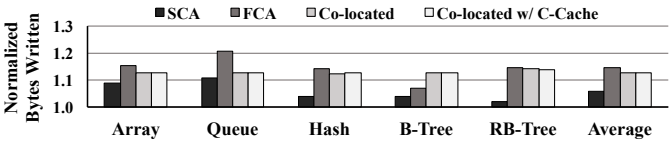


Figure 14: Write Traffic to NVMM normalized to the *no-encryption* design (lower is better).

Reducing the write traffic not only provides better performance, but also improves the lifetime of NVM. Selective counter-atomicity can improve the NVMM lifetime by 6.6% assuming a uniform wear-leveling technique [38] (an orthogonal design consideration in the NVMM systems). The improvement will be higher if we consider compressing the counters using techniques proposed by some prior works [1, 36].

6.3.4. Sensitivity to Counter Cache Size. Figure 15 compares the performance of the *selective counter-atomicity* design when we vary the counter cache size from 128KB to 8MB and run workloads with footprints ranging from 100MB to 1000MB. Figure 15(a) and 15(b) show the average speedup and miss rate with different counter cache sizes over the smallest 128KB counter cache. We observe that as the size of the counter 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 8MB counter cache improves the performance by 9% over a 128KB counter cache when the workloads have 100MB footprint. On the other hand, the improvement is only 2.4% with 1000MB workloads. Similarly, using an 8MB counter cache decreases the miss rate by 23.3% with a 100MB workload, while the miss rate decreases by 15.4% with a 1000MB 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 NVMM encryption work [3].

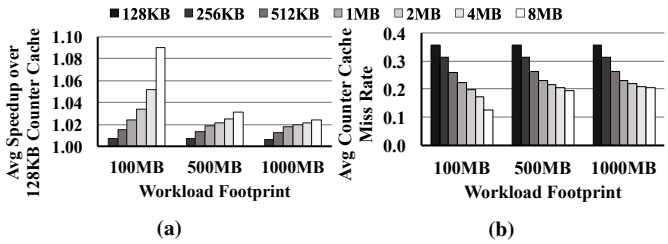


Figure 15: Evaluating SCA with different sizes of counter cache. (a) Average speedup over a 128KB counter cache (higher is better). (b) Average counter cache miss rate (lower is better).

6.3.5. Sensitivity to Transaction Size. In this experiment, we evaluate the overhead of *selective counter-atomicity* with variable transaction size. Figure 16 compares the performance of *selective counter-atomicity* when varying the transaction size from 64B to 4KB. 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 (4KB). 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 NVMM applications that manipulate a large dataset within a transaction.

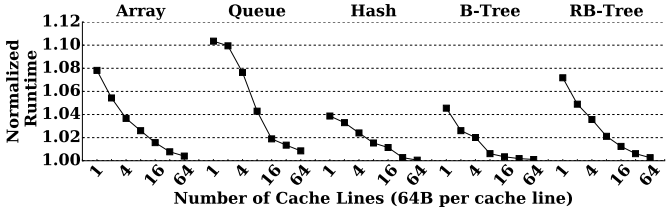
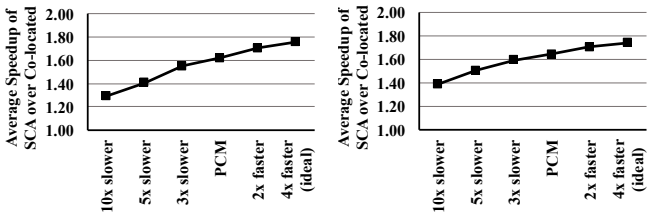


Figure 16: The runtime of SCA with different sizes of transaction, normalized to the ideal design (lower is better).

6.3.6. Sensitivity to NVM latency. In this experiment, we compare the performance of *selective counter-atomicity* with

the design that co-locates both the counter and data (Section 3.2.1) with varying NVM latency (Figure 17) to understand the performance sensitivity of *selective counter-atomicity* to different NVM 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 17(a) 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 17(b) 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 NVM 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 NVM for lower latency and higher throughput, hereby adopting *selective counter-atomicity* will lead to better performance.



(a) Read Latency **(b) Write Latency**

Figure 17: Varying (a) read latency, and (b) write latency.

6.3.7. 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 [3, 47, 59]. The size of our counter cache is 1MB per core, similar to the one employed by Awad et al. [3]. Our proposed design, *selective counter-atomicity*, requires only an additional 16-entry (1kB in size) *counter write queue* at the memory controller.

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 [2, 3, 5, 6, 11, 13, 17, 24, 25, 26, 27, 29, 30, 34, 37, 45, 53, 55, 57, 65]. The true potential of a persistent memory can be exploited when the NVM is on the memory bus with a low access latency. In these cases, the encryption latency (40 ns) 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 [47, 61]. However, the commodity NVM chips are yet to become commercialized in a wide scale and the current NVM products still place NVM over the PCIe bus with a latency close to high-end SSDs [16, 18, 42, 49, 50]. 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 NVM technologies over time and harness its true potential by placing NVM on the memory bus. Our proposed technique to optimize *counter-atomicity* would be highly valuable in those systems.

8. Related Work

To the best of our knowledge, this is the first work to study *crash consistency in the encrypted NVMM systems*. In this section, we discuss relevant studies on NVM encryption and crash consistency. We classify the related works into five broad categories.

NVM Encryption. Memory encryption has been a research area for decades [44, 47, 48, 60]. Recent studies on NVM encryption focus on providing faster and efficient encryption techniques [3, 8, 61]. DEUCE reduces the write traffic using a dual-counter encryption mechanism to improve the lifespan of the encrypted NVMs [61]. Another recent work, Silent Shredder, proposes to eliminate writes when writes are just zeroing out physical pages to achieve better power consumption and performance [3]. i-NVMM provides an encryption technique for NVM that leaves hot data unencrypted until a power failure is detected such that it reduces the en/decryption overhead during the normal execution time [8]. These proposals *do not* consider crash consistency in encrypted NVMs and therefore, are orthogonal to this paper. *Counter-atomicity* has to be applied on top of these efficient encryption mechanisms in order to guarantee crash consistency in the encrypted NVMM systems.

Memory Persistency. The memory persistency models prescribe the order of writes to NVMM to ensure the recoverability of in-memory persistent data in a consistent state. Pelly et al. propose the notion of persistency [37], and later works propose different relaxed persistency models to allow higher concurrency in memory writes [27, 34]. Recently, Intel extended its ISA to support persistency and released an NVM library to manipulate the low-level persistency support [19, 20]. Our proposed *counter-atomicity* is orthogonal to any *memory persistency model*. It does not enforce the ordering of writes to persistent memory, but only enforces that the data and counter of the same write request are persisted at the same time. Under this assumption, crash consistent programming models can be extended to the encrypted NVM and *counter-atomicity* can be applied on top of any persistency model to ensure consistent data recovery across system failures.

Software-Based Crash Consistency. NV-Heaps [11], Mnemosyne [53], NVML [20], DudeTM [30], LSNVMM [17]

and REWIND [5] provide various types of transactional interfaces for NVMM to ease the burden of managing persistent data in a crash consistent manner. BPFS [12], SCMFS [56], PMFS [15], Aerie [52] and Mojim [64] provide crash consistent file systems to leverage the performance benefit of NVMs to store persistent data. All these techniques will have to adopt *counter-atomicity* to extend to an encrypted NVMM system.

Hardware-Assisted Transaction. ThyNVM [39] and ATOM [24] provide hardware-based transactional interfaces for NVMM, without relying on `clwb` or `sfence` instructions to maintain the correct ordering of writes. Instead, they adopt mechanisms like logging and checkpointing in the hardware. Some prior studies also propose to use non-volatile caches (NV-cache) to solve crash consistency issues (e.g., Kiln [65]). All these techniques also need to follow *counter-atomicity* to provide crash consistency in an encrypted NVM.

Battery-Backed NVMM Systems. Backing up the entire memory after a power failure can ensure crash consistency in an NVMM system. For example, whole system persistence (WSP) proposes to use residual energy to make sure that the volatile data is written back to NVMM after a power failure [35]. i-NVMM also relies on a battery to encrypt and persist a fraction of volatile data at a power failure [8]. *Counter-atomicity* is not required in these techniques as the entire system is backed up. However, our mechanism does not need a battery to backup the *entire* system. Our solution extends the ADR to cover the *counter write queue*, such that both the *data and counter write queues* drain after a failure. As the size of the write queue is small, our solution is adaptable to other systems.

9. Conclusion

This is the first work that introduces *counter-atomicity*, a requirement that guarantees the recovery of encrypted in-memory persistent data in a consistent state across system failures. *Counter-atomicity* ensures that both data and the associated counter values that are required for correct decryption are persisted in memory *atomically*. Enforcing *counter-atomicity* for *all* writes to NVMM results in a significant performance degradation. Our *selective counter-atomicity* approach takes advantage of the writes that *do not* immediately affect the consistent recoverable states and allows reordering and caching of the counter and data writes to reduce the performance overhead. We provide simple primitives to let the programmers selectively enforce *counter-atomicity* in the persistent applications and propose a low-overhead hardware design to enforce *counter-atomicity* in an encrypted and crash consistent NVMM system. We believe that our work provides a holistic system support for both data persistence and security, paving the path for a wide-spread adoption of NVMM systems.

10. Acknowledgements

We would like to thank the reviewers, Marzieh Lenjani and Yizhou Wei for their valuable feedback. This work was supported by the National Science Foundation under the award 1566483.

References

[1] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *ISCA*, 2004.

[2] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, 2015.

[3] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne. Silent Shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *ASPLOS*, 2016.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[5] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5), 2015.

[6] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu. Efficient support of position independence on non-volatile memory. In *MICRO*, 2017.

[7] S. Chen and Q. Jin. Persistent B+-Trees in non-volatile main memory. In *VLDB*, 2015.

[8] S. Chhabra and Y. Solihin. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *ISCA*, 2011.

[9] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *SOSP*, 2013.

[10] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *SOSP*, 2013.

[11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS*, 2011.

[12] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[13] Z. Deng, L. Zhang, N. Mishra, H. Hoffmann, and F. T. Chong. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in NVMs. In *MICRO*, 2017.

[14] W. Diffie and M. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3), 1979.

[15] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.

[16] Everspin Technologies. Storage solutions achieve greater performance with MRAM. https://www.everspin.com.

[17] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda. Log-structured non-volatile main memory. In *ATC*, 2017.

[18] 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.

[19] Intel Corporation. Intel architecture instruction set extensions programming reference (319433-029 April 2017). https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[20] Intel Corporation. Persistent memory programming. http://pmem.io/.

[21] Intel Corporation. Platform brief Intel Xeon processor c5500/c3500 series and Intel 3420 chipset. https://www.intel.com/content/www/us/en/intelligent-systems/picket-post/embedded-intel-xeon-c5500-processor-series-with-intel-3420-chipset.html.

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

[23] 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.

[24] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *HPCA*, 2017.

[25] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. Language-level persistency. In *ISCA*, 2017.

[26] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch. High-performance transactions for persistent memories. In *ASPLOS*, 2016.

[27] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. In *MICRO*, 2016.

[28] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *ISPASS*, 2013.

[29] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[30] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, and J. Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *ASPLOS*, 2017.

[31] K. Ma, X. Li, J. Li, Y. Liu, Y. Xie, J. Sampson, M. T. Kandemir, and V. Narayanan. Incidental computing on IoT nonvolatile processors. In *MICRO*, 2017.

[32] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1), 1992.

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

[34] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with WHISPER. In *ASPLOS*, 2017.

[35] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS*, 2012.

[36] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *PACT*, 2012.

[37] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *ISCA*, 2014.

[38] M. K. Qureshi, M. Franchescini, V. Srinivasan, L. Lastras, B. Abali, and J. Karidis. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *MICRO*, 2009.

[39] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *MICRO*, 2015.

[40] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: Challenges in and avenues for CMP scaling. In *ISCA*, 2009.

[41] A. M. Rudoff. Deprecating the pcommit instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, 2016.

[42] A. 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.

[43] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable SSD. In *OSDI*, 2014.

[44] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *ISCA*, 2005.

[45] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *MICRO*, 2017.

[46] Storage Networking Industry Initiative (SNIA). NVDIMM messaging and FAQ. https://www.snia.org/sites/default/files/NVDIMM%20Messaging%20and%20FAQ%20Jan%2020143.pdf.

[47] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *MICRO*, 2003.

[48] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *ISCA*, 2005.

[49] B. 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.

[50] G. M. 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.

[51] 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 *SC*, 2014.

[52] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*, 2014.

[53] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memeory. In *ASPLOS*, 2011.

[54] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. In *VLDB*, 2014.

[55] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck. Hardware supported persistent object address translation. In *MICRO*, 2017.

[56] X. Wu and A. L. N. Reddy. SCMFS: A file system for storage class memory. In *SC*, 2011.

[57] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie. Overcoming the challenges of crossbar resistive memory architectures. In *HPCA*, 2015.

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

[59] C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *ISCA*, 2006.

[60] J. Yang, Y. Zhang, and L. Gao. Fast secure processor for inhibiting software piracy and tampering. In *MICRO*, 2003.

[61] V. Young, P. J. Nair, and M. K. Qureshi. DEUCE: Write-efficient encryption for non-volatile memories. In *ASPLOS*, 2015.

[62] J. Zhan, O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie. OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures. In *MICRO*, 2016.

[63] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *MSST*, 2015.

[64] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *ASPLOS*, 2015.

[65] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, 2013.