A Secure and Persistent Memory System for Non-volatile Memory

Pengfei Zuo[*] [†], Yu Hua[*], Yuan Xie[†]
[*]*Huazhong University of Science and Technology*
[†]*University of California, Santa Barbara*

Abstract

In the non-volatile memory, ensuring the security and correctness of persistent data is fundamental. However, the security and persistence issues are usually studied independently in existing work. To achieve both data security and persistence, simply combining existing persistence schemes with memory encryption is inefficient due to crash inconsistency and significant performance degradation. To bridge the gap between security and persistence, this paper proposes **SecPM**, a **Sec**ure and **P**ersistent **M**emory system, which consists of a counter cache write-through (CWT) scheme and a locality-aware counter write reduction (CWR) scheme. Specifically, SecPM leverages the CWT scheme to guarantee the crash consistency via ensuring both the data and its counter are durable before the data flush completes, and leverages the CWR scheme to improve the system performance via exploiting the spatial locality of counter storage, log and data writes. We have implemented SecPM in gem5 with NVMain and evaluated it using five widely-used workloads. Extensive experimental results demonstrate that SecPM reduces up to half of write requests and speeds up the transaction execution by $1.3\times \sim 2.0\times$ via using the CWR scheme, and achieves the performance close to an un-encrypted persistent memory system for large transactions.

1 Introduction

As DRAM suffers from limited scalability and high power leakage [30, 45], non-volatile memories (NVM), such as PCM [48], ReRAM [1], STT-RAM [2], and 3D XPoint [17], become promising candidates of the next-generation main memory. NVM has the advantages of high scalability, high density, and near-zero standby power. However, two fundamental issues need to be addressed in order to effectively use NVM in memory systems, i.e., data persistence and security.

• ***Persistence Issue.*** The non-volatility of NVM enables data to be persistently stored into main memory for instantaneous failure recovery. In order to ensure the correctness of persistent data, crash consistency guarantee is fundamental [32, 47], which needs to achieve the correct recovery of persistent data in case of a system failure, e.g., power failure and system crash. Specifically, NVM systems typically contain volatile storage components, e.g., CPU caches and possible DRAM. If a system failure occurs when a data structure in NVM is being updated, the data structure may be left in a corrupted state. Moreover, modern processor and memory controller usually reorder memory writes. The partial update and reordering cause the crash inconsistency in NVM [26, 53]. Hence, cache line flush, memory barrier, and log-based mechanisms are used to ensure the crash consistency [28, 50].

• ***Security Issue.*** The non-volatility of NVM also causes the security problem of data remanence vulnerability [3, 55], since NVM still retains data after systems are powered down. In general, when using encryption to protect data security, the encrypted data are stored in disks, while raw data are retained in main memory [12]. In the legacy DRAM main memory, if a DRAM DIMM is stolen, data are quickly lost due to the volatility. Unlike it, if an NVM DIMM is stolen, an attacker can directly stream out the data from the DIMM. Hence, memory encryption becomes important to ensure the data security in NVM. Counter mode encryption [27, 55] is usually used in secure NVM, due to the low decryption latency and high security level.

However, existing schemes addressing the persistence issue [22, 32, 38, 47, 50] usually fail to efficiently use memory encryption in NVM systems. On the other hand, existing schemes addressing the security issue [3, 7, 44, 55, 59] are inefficient to guarantee the data consistency. To achieve both data persistence and security in NVM, simply combining existing persistence schemes with memory encryption does not work due to the following challenges.

• ***Consistency Challenge.*** In order to guarantee crash consistency, it is essential to use cache line flush and memory barrier instructions to persist data into NVM with correct ordering. In the counter mode encryption, each memory line is encrypted and decrypted using a counter, and the counter increases one on each write [27]. Thus each write in the secure NVM has to persist two data including the data itself and its counter [29]. The data is evicted from the CPU caches and the counter is evicted from the counter cache managed by the memory controller. The two writes have to be simultaneously persisted to ensure that the persistent data can be correctly decrypted across system failures. For example, if the system fails after the encrypted data is persisted into NVM, but before its counter has been persisted, the persisted data cannot be decrypted without the correct counter during the recovery from a system failure.

However, current computer systems fail to simultaneously persist the two writes, thus resulting in an inconsistent state on a system failure [29, 43]. This is because cache line flush and memory barrier instructions only ensure the data from the CPU caches to be correctly persisted into NVM, which fail to operate the counter cache and hence cannot ensure the crash consistency of counters.

- ***Performance Challenge.*** Each data write in the secure NVM generates two NVM write requests (one writes the data and the other writes its counter), which significantly degrades the system performance. Because writes to NVM usually incur much higher latency than reads (i.e., $3 \sim 8\times$) [35, 57], and are in the critical path of application execution due to persistence requirements [46, 47]. Moreover, more write requests also significantly increase the latency of read requests to the same bank. When a write request is served by an NVM bank, the following read and write requests to the same bank are blocked and have to wait until the current write request is completed [34].

To bridge the gap between security and persistence, this paper proposes a secure and persistent memory system, called SecPM. SecPM proposes to use a simple yet effective counter cache write-through (CWT) scheme with a register to ensure the crash consistency of both data and counters. CWT appends the counter of the data in the write queue during encrypting the data, which ensures the counter is durable before the data flush completes. Thus the two writes for the data and its counter are performed in an atomic manner. Moreover, SecPM leverages a locality-aware counter write reduction (CWR) scheme to improve the system performance. CWR explores and exploits the spatial locality of counter storage, log and data writes to merge write requests from counters in the write queue, thus significantly reducing the number of NVM write requests. In summary, this paper makes the following contributions:

- ***Simple yet Effective Consistency Guarantee.*** Existing work adds new primitives in the programming language to explicitly flush programmer-specified counter cache lines in the counter cache for consistency guarantee [29], which however incurs modifications on both hardware and software layers. The novelty of SecPM is to leverage a simple yet effective counter cache write-through (CWT) scheme with a register to guarantee the crash consistency of secure NVM. Our scheme is able to ensure both data and counters to be simultaneously persisted before a CPU cache line flush instruction is retired.

- ***Significant Performance Improvement.*** SecPM leverages a locality-aware counter write reduction (CWR) scheme to improve system performance via significantly reducing the number of NVM write requests from counters. Extensive experimental results demonstrate that SecPM reduces up to half of write requests, and speeds up the transaction execution by $1.3 \sim 2.0$ times by using the CWR scheme, and achieves the performance close to an un-encrypted persistent memory system for large transactions.

- ***Programmer-transparent Implementation.*** The implementation of SecPM only needs to perform slight modifications on the hardware layer without any modifications on the software layer, e.g., programming language and compiler, which are transparent for programmers and applications. Thus the programs and applications running on an un-encrypted NVM can be directly executed on a secure NVM with SecPM.

2 Background and Motivation

2.1 Consistency Guarantee for Persistence

In order to correctly persist data into NVM, it is important to guarantee the crash consistency. Durable transaction [22, 28] is a commonly used solution for crash consistency guarantee, which enables a group of memory updates to be performed in an atomic manner. Figure 1 shows the pseudo-code that an application implements a durable transaction with undo logging, which include three stages, i.e., prepare, mutate and commit. Specifically, the prepare stage creates a log entry to back up the data to be written; the mutate stage modifies the data in place; the commit stage invalidates the log entry created in the prepare stage. Whichever stage a system failure occurs, the application can be recoverable in a consistent state since at least one of the log and data are consistent.

As modern CPU and memory controller usually reorder memory writes, the durable transaction uses cache line flush and memory barrier instructions to enforce write ordering [22, 28, 32]. The flush instructions including `clflush`, `clflushopt`, and `clwb` explicitly flush a dirty CPU cache line into the write queue of the memory controller. The memory barrier instructions including `mfence` and `sfence` order the memory operations via blocking the memory operations after the fence, until the memory operations before the fence complete. The `pcommit` instruction was initially used to force the write requests

```
DurableTx(data, log) {
    do some computation;
    // Prepare stage: backing up the data in log
    write undo log;
    flush log;
    memory_barrier();
    // Mutate stage: updating the data in place
    write data;
    flush data;
    memory_barrier();
    // Commit stage: invalidating the log
    log->valid = false;
    flush log->valid;
    memory_barrier();
}
```

Figure 1: Steps implementing a durable transaction.

in the write queue into NVM but was deprecated later by Intel [18], due to the use of asynchronous DRAM refresh (ADR) mechanism [19, 29, 41]. ADR is able to persist the write requests in the write queue into NVM in case of a system failure via the battery backup. Therefore, the cache lines reaching the write queue are considered durable.

2.2 Memory Encryption for Security

Since NVM suffers from the data remanence vulnerability, memory encryption is non-trivial for NVM to ensure data security.

2.2.1 Security Guarantee of Encryption

A straightforward method to encrypt a memory line is to use a block cipher algorithm, e.g., AES [11], with a global key, as shown in Figure 2a. However, an attacker can know which lines have the same content via simply comparing encrypted lines, since all lines are encrypted using the same key, which is vulnerable to dictionary and replay attacks [3, 55]. Using the key along with the line address to encrypt each line is a more secure method, as shown in Figure 2b, which can ensure that different lines are encrypted with different keys. However, this method is still vulnerable to the dictionary attack for a single line, if an attacker monitors consecutive writes to this line.

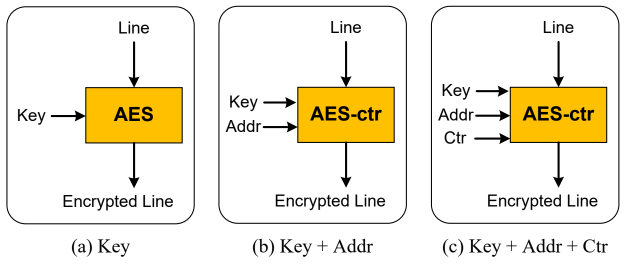


Figure 2: The encryption methods (a) using a global key; (b) using a key and line address; (c) using a key, line address and a counter (ctr).

A secure method is to encrypt each line by using the key and line address in conjunction with a per-line counter, as shown in Figure 2c. The counter increases by one on each write and hence consecutive writes to the same line are encrypted with different keys, achieving high security level.

2.2.2 Latency Reduction via One Time Pad

As memory reads are in the critical path of program execution, memory encryption causes the high decryption latency after each memory read due to serial execution, as shown in Figure 3a, thus significantly degrading the system performance. Counter mode encryption [27] is hence proposed to reduce the decryption latency from the critical path of memory reads via leveraging the One Time Pad (OTP) technique, and hence has been widely used in

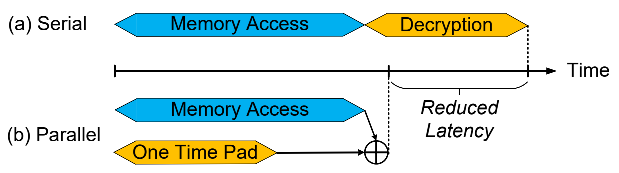


Figure 3: Reducing the decryption latency using One Time Pad (OTP).

encrypted memory systems [3, 7, 44, 55, 59]. The main idea is to compute an OTP in parallel with a memory read, and then XOR the OTP with the ciphertext data to generate the plaintext, thus hiding the decryption latency in the memory access latency, as shown in Figure 3b.

2.2.3 Operations of Counter Mode Encryption

The security of counter mode encryption is based on the premise that each OTP is never reused for data encryption [27, 44, 55, 59]. To ensure this, the counter mode encryption uses a secret key, the line address and the per-line counter to generate the OTP through the AES circuit, as shown in Figure 4. For a memory write, the cache line to be written is encrypted by XORing its content with the OTP. To read a memory line, we decrypt it by XORing its content with the OTP. All counters are retained in main memory. In order to reduce the generation time of OTPs, the memory controller manages an on-chip counter cache to buffer the recently-accessed counters [29, 44, 59].

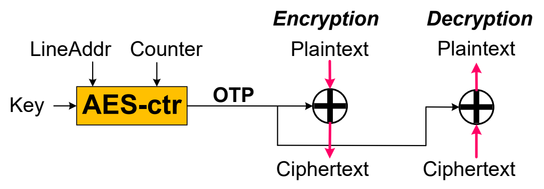


Figure 4: The encryption and decryption process in counter mode encrpytion.

2.3 The Gap between Persistence & Security

Each cache line flushed from CPU caches in the encrypted NVM produces two write requests: one for the data and the other for the counter. This characteristic not only degrades the system performance since NVM needs to deal with more write requests, but also incurs the crash inconsistency problem. The reason is that the data is evicted from the CPU caches but its counter is evicted from the counter cache. However, the cache line flush instruction only flushes the data in CPU caches, but fails to operate the counter cache. The memory barrier instruction only ensures the ordering of the CPU cache line flushes, but cannot work for the counter writes [29, 43]. As a result, the data and its counter cannot be simultaneously persisted, and thus the persisted data cannot be decrypted without correct counters during the recovery from a system failure.

Table 1: The recoverability when a system failure occurs in the different stages of a transaction.

Stage	Log Content	Log Counter	Data Content	Data Counter	Recoverable ?
Prepare	Wrong	Wrong	Correct	Correct	Yes
Mutate	Correct	**Unknown**	Wrong	Wrong	**No**
Commit	Correct	**Unknown**	Correct	**Unknown**	**No**

We analyze the recoverability when a system failure occurs in the different stages of a durable transaction executed in an encrypted NVM, as shown in Table 1. We observe when a system failure occurs in the mutate and commit stages, the data are unrecoverable. Specifically, when a system failure occurs in the prepare stage, the data contents and the counters encrypting the data are unmodified and correct, which are in a consistent state. However, when a system failure occurs in the mutate stage, the data are not updated completely and become wrong. The contents of the log are correct due to the use of log flush and memory barrier instructions, but whether the counters encrypting the log are correctly persisted is unknown, since the cache line flush and memory barrier instructions fail to operate the counters stored in the counter cache. Hence, during a recovery, the log cannot be decrypted due to no correct counters, thus failing to recover the logged data. For the same reasons, when a system failure occurs in the commit stage, the correctness of both log and data counters are unknown, and hence the data are unrecoverable.

To address the inconsistency problem in the encrypted NVM, Liu et al. [29] proposed the novel concept of the selective counter-atomicity for the first time, which indicates that either both data and its associated counter have been simultaneously persisted or not. To efficiently implement the counter-atomicity, software and hardware layers need to be modified [29]. First, two new primitives including `CounterAtomic` variable and `counter_cache_writeback()` function, are added in the programming language to explicitly flush programmer-specified counter cache lines. Second, the compiler is modified to support the new primitives. Third, a counter write queue is added into the memory controller to schedule the counter and data writes. As a result, the programs initially running on a system with the un-encrypted NVM cannot directly run on a system with the secure NVM. Our work aims for a different design goal. The proposed SecPM is a programmer-transparent solution to guarantee the crash consistency of the secure NVM without the needs of modifications on programming language and compiler, which is detailed in the rest of the paper.

3 The SecPM Design

3.1 Threat Model

Our threat model is similar to existing work on secure NVM [3, 29, 44, 55, 59], which aims to protect NVM from two well-known physical access based attacks, including

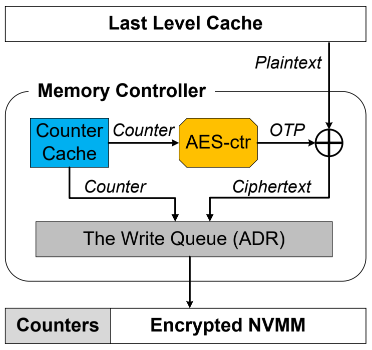


Figure 5: The hardware architecture of SecPM. *(The asynchronous DRAM refresh (ADR) mechanism is able to persist write requests in the write queue into NVMM on a system failure via the battery backup. Therefore, the cache lines reaching the write queue are considered durable.)*

stolen DIMM and bus snooping attacks. In the stolen DIMM attack, since NVM still retains data after systems are powered down, an attacker can easily stream out the data stored in the NVM after stealing the NVM DIMM. In the bus snooping attack, since NVM is accessed through the memory bus, an attacker can insert a bus snooper to obtain the data through the bus. We do not consider bus tampering attacks in the threat model like existing work [3, 29, 44, 55, 59]. These attacks can be defended via Merkle Trees based authentication techniques [42, 54], which are orthogonal to our work.

3.2 An Architectural Overview

To bridge the gap between security and persistence in NVM, we propose SecPM, a **Sec**ure **P**ersistent **M**emory system, which leverages a simple yet effective counter cache write-through scheme (§3.3) with a register to guarantee the crash consistency of both data and its counter, and a counter write reduction scheme (§3.4) to significantly reduce the number of write requests for improving the system performance. Moreover, SecPM guarantees the consistency of page re-encryption that handles counter overflow by reusing the ADR mechanism (§3.5).

The hardware architecture of SecPM is shown in Figure 5. When CPU issues a flush instruction, the corresponding cache line is evicted from the last level cache to the memory controller. The memory controller encrypts the cache line using a counter and then appends the encrypted cache line in the write queue. The proposed counter cache write-through (CWT) scheme is performed in the counter cache, and the counter write reduction (CWR) scheme is performed in the write queue. As a whole, SecPM only performs slight hardware modifications on the memory controller, which are transparent for programmers and applications. Thus programs and applications running on an un-encrypted NVM can be directly executed on a secure NVM with SecPM.

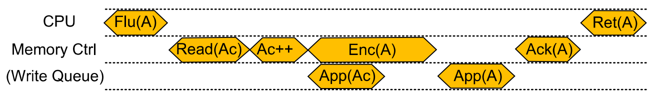


Figure 6: The sequence that the memory controller deals with a cache line flush by using the counter cache write-through scheme. *(**Flu(A)**: flushing the cache line **A** into NVM; **Ac**: the counter of **A**; **App(Ac)**: appending **Ac** in the write queue; **Enc(A)**: encrypting **A**; **Ret(A)**: the flush of **A** is retired.)*

3.3 Crash Consistency Guarantee

3.3.1 Counter Cache Write Through Scheme

SecPM employs a counter cache write-through (CWT) scheme in the counter cache, which writes each dirty counter in the counter cache, and simultaneously writes the counter copy in the write queue. We further show how to schedule the cache line flush via using the simple CWT scheme to ensure the crash consistency of the secure NVM.

Figure 6 shows the sequence diagram that the memory controller deals with a cache line flush in SecPM. When the CPU issues a flush for cache line *A* (*Flu(A)*), the memory controller reads the counter of *A* from the counter cache (*Read(Ac)*), and then adds the counter by one (*Ac*++). The updated counter is used to encrypt *A* (*Enc(A)*). During the encryption, the updated counter is written back to the counter cache, and simultaneously appended in the write queue (*App(Ac)*) via the CWT scheme. After the encrypted *A* is appended in the write queue (*App(A)*), the memory controller sends an ack (*Ack(A)*) to the CPU, and the cache line flush is retired (*Ret(A)*). From Figure 6, we observe that the counter encrypting a CPU cache line has been already appended in the write queue before the cache line flush completes via the CWT scheme. Hence, if the contents of log and data have been persisted and become correct, their counters have also been persisted.

Table 2: The recoverability when a system failure occurs in the different stages of a transaction in SecPM.

Stage	Log Content	Log Counter	Data Content	Data Counter	Recoverable ?
Prepare	Wrong	Wrong	Correct	Correct	Yes
Mutate	Correct	Correct	Wrong	Wrong	Yes
Commit	Correct	Correct	Correct	Correct	Yes

Table 2 shows the recoverability when a system failure occurs in the different stages of executing a durable transaction in SecPM. In the prepare stage, the contents and counters of data are un-modified and thus correct. When the prepare stage completes, it means that all contents of the log have been successfully flushed and persisted. As we demonstrate above, if the contents of log have been persisted, their counters have also been persisted by using the CWT scheme. Hence, both the contents and counters of the log have been persisted and thus are correct in the mutate stage. For the same reason, in the commit stage, the contents and counters of the log and data are correct. We observe that at least one of log and data are correct whichever stage a system failure occurs in, and hence the system can be recoverable in a consistent state.

Recovery: After a system failure occurs, the recovery scheme of transactions in SecPM is the same as that in an un-encrypted persistent memory. During the recovery, the application scans the log region, and checks each log entry. For each log entry, the application checks whether the log is complete via the log-end tag of a transaction. If the log is incomplete, it means the system crashes in the prepare stage of the transaction, in which the original data are correct. The application directly abandons the incomplete log. If the log is complete, it means the system crashes in the mutate or commit stage, in which the log is correct. The application undoes the data via the log. Therefore, the application can be recovered in a consistent state in SecPM.

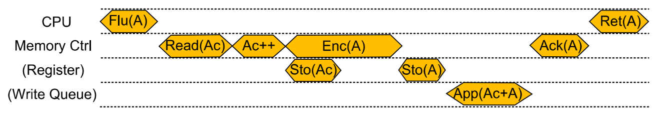


Figure 7: The sequence that the memory controller deals with a cache line flush by using the counter cache write-through scheme with a register. *(**Sto(Ac)**: storing **Ac** in the register.)*

3.3.2 A Register for Atomic Write Consistency

In addition to using durable transaction, some existing work, e.g., wB$^+$-tree [5], WORT [26], FAST&FAIR [16], and level hashing [58], exploits the atomic write of NVM to ensure the crash consistency of data structures, thus avoiding the overhead of logging or copy-on-write mechanisms. However, simply performing the proposed counter cache write-through (CWT) scheme cannot ensure the crash consistency of the atomic-write data. Specifically, we consider that the flush for cache line *A* as shown in Figure 6 is an atomic write, without the data backup based on logging. If a system failure occurs after appending the counter of *A* (*App(Ac)*) before appending *A* (*App(A)*) in the write queue, the counter of *A* (A_c) is updated and persisted in NVM using the asynchronous DRAM refresh (ADR) mechanism but *A* is not. After the system is recovered from the failure, the old value of *A* cannot be decrypted using the new counter. *A* is an atomic write without data backup, thus resulting in an inconsistent state.

To guarantee the crash consistency of atomic writes in the encrypted NVM, we add a register for each AES circuit. During encrypting the data (i.e., a cache line) evicted from CPU caches, we store its corresponding counter in the register ($Sto(Ac)$) instead of directly appending the counter in the write queue, as shown in Figure 7. After finishing

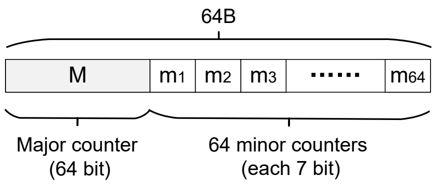


Figure 8: The counter storage of memory lines in a physical page. *(A 4KB page includes 64 memory lines. Each memory line is encrypted using the major counter concatenated with a minor counter.)*

encrypting the data, we first store the encrypted data in the register ($Sto(Ac)$), and then simultaneously append the encrypted data and its counter in the write queue ($App(Ac+A)$). We observe that both data and its associated counter simultaneously exist in the write queue or not, by using a register. As a result, the crash consistency of atomic writes is ensured in SecPM. Moreover, since the size of the register is very small, i.e., 2 cache lines (one for the data cache line and the other for its counter cache line), and reads and writes in such a small register are very fast, the use of the register has a negligible impact on the system performance.

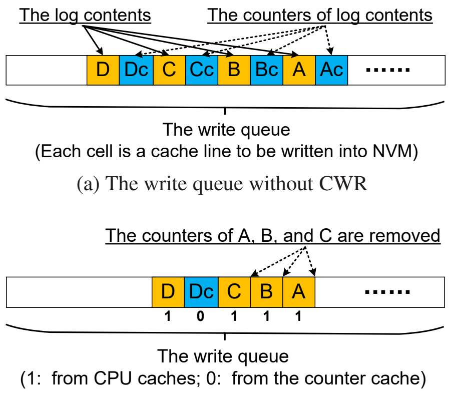
3.4 Counter Write Reduction

In the secure NVM, each CPU cache line flush appends two write requests in the write queue, which doubles the number of write requests, compared with an un-encrypted NVM. Thus the performance of the memory system would significantly decrease. SecPM proposes a locality-aware counter write reduction (CWR) scheme to improve the system performance via leveraging the spatial locality of counter storage, log and data writes.

3.4.1 Spatial Locality of Counter Storage

In order to reduce the storage overhead of counters, the counter mode encryption uses a shared major counter (M) for an entire page and 64 minor counters (m_1, m_2, ..., m_{64}) each for a memory line in the 4KB page [3, 52], as shown in Figure 8. The major counter is 64 bits and each minor counter is 7 bits. Thus the counters of all memory lines in a page are 64B and stored in one memory line, exhibiting good spatial locality.

In a page, each memory line is encrypted by the per-page major counter concatenated with a per-line minor counter. When a memory line is rewritten, its corresponding minor counter increases by one. Although updating a minor counter only modifies several bits, persisting the minor counter has to write the entire memory line into NVM since a memory line is the basic unit of memory writes. When a minor counter overflows, counter mode encryption increases the major counter by one, resets all minor counters to 0, and re-encrypts all memory lines in the page using the new counters [52]. The process of re-encrypting a page is presented in Section 3.5. The 64-bit major counter cannot


The log contents
The counters of log contents
D
Dc
C
Cc
B
Bc
A
Ac
......
The write queue
(Each cell is a cache line to be written into NVM)
The counters of A, B, and C are removed
D
Dc
C
B
A
......
1
0
1
1
1
The write queue
(1: from CPU caches; 0: from the counter cache)

(a) The write queue without CWR

(b) The write queue with CWR

Figure 9: An illustration of the write queue when writing a log with and without CWR scheme.

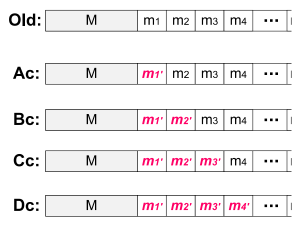
overflow throughout the lifespan of an NVM, since the count range, i.e., $2^{64} \approx 10^{20}$, is far larger than the cell endurance limit of NVM, e.g., $10^7 \sim 10^9$ for PCM [35, 57] and $10^8 \sim 10^{12}$ for ReRAM [24, 25].

3.4.2 Spatial Locality of Log and Data Writes

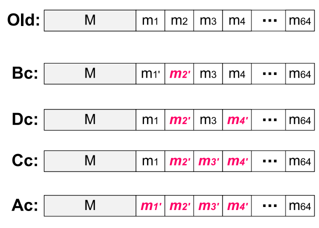
Since a log is stored in a contiguous region in NVM, the log writes of a transaction flush multiple cache lines which have the contiguous physical addresses, thus having good spatial locality. Moreover, the data writes of a transaction usually have good spatial locality, since programs usually allocate a contiguous memory region for a transaction [15]. Hence, the cache lines flushed into the contiguous region have the contiguous physical addresses. For example, a transaction inserts a 1KB key-value item into a key-value store maintained in NVM, which will flush 16 cache lines with the contiguous physical addresses.

3.4.3 Counter Write Reduction Scheme

Based on the locality of counter storage, log and data writes, we show the write queue during flushing the log entry of a transaction in Figure 9a. The log entry contains multiple cache lines (i.e., A, B, C, and D) with contiguous physical addresses in the same physical page. Since all counters of a page are contained in one memory line as shown in Figure 8, the counter cache lines of the log entry, i.e., A_c, B_c, C_c, and D_c, will be written to the same memory line. Moreover, since these counter cache lines are evicted from the write-through counter cache, the latter counter cache lines contain the updated contents of the former ones with the same address. For example, the memory lines A, B, C and D correspond to the minor counters m_1, m_2, m_3 and m_4, respectively. A_c, B_c, C_c, and D_c are written into NVM in order, as shown in Figure 10a. We observe that the counter cache line A_c only contains the updated minor counter $m_1{}'$, and the counter cache line B_c contains the updated minor counters $m_1{}'$ and $m_2{}'$, and the counter cache line C_c contains the updated minor counters $m_1{}'$, $m_2{}'$ and $m_3{}'$. Moreover, the



(a) In-order writes



(b) Out-of-order writes

Figure 10: The contents of the corresponding counter cache lines when the CPU cache lines A, B, C and D are flushed in order or out of order.

multiple cache lines of a log entry, i.e., A, B, C and D, may be flushed from CPU caches out of order. For example, B, D, C and A are flushed in turn. Thus the corresponding counter cache lines are written into the write queue in the order of B_c, D_c, C_c and A_c. In this case, it is still valid that the counter cache lines written latter contain the updated contents of the former ones, as shown in Figure 10b.

According to above observations and insights, we present a counter write reduction (CWR) scheme. Specifically, when a new counter cache line evicted from the counter cache reaches the write queue, we check whether a counter cache line in the write queue has the same physical address as the new one. If yes, we remove this existing counter cache line without causing any loss of data, since the new counter cache line contains the updated contents of the removed one as shown in Figure 10. To reduce the latency of checking the cache lines with the same address, we add a one-bit flag for each cache line in the write queue. The flag is used to distinguish whether a cache line is from CPU caches or the counter cache. , e.g., '1' for the cache lines from CPU caches and '0' for those from the counter cache. Thus we can check the cache lines only from the counter cache based on the flag. By performing the CWR scheme, the new write queue is shown in Figure 9b. We observe that the number of write requests is significantly reduced, since the counters of A, B and C are removed. When a transaction flushes a log with the size of one page, $64*2=128$ CPU and counter cache lines are written into NVM without our proposed CWR scheme. By using CWR, only $64+1=65$ cache lines are written, reducing almost half of NVM writes.

3.5 Page Re-encryption to Handle Overflow

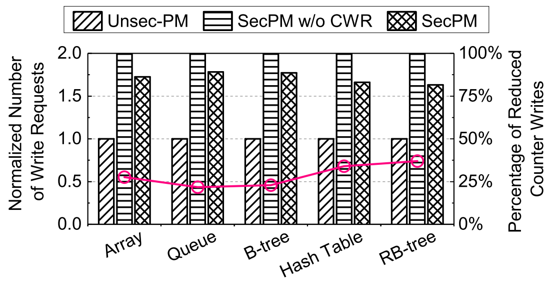
In the counter mode encryption, the major counter cannot overflow, as discussed in Section 3.4.1. When the minor counter of a memory line in a page overflows, the page needs to be re-encrypted using the updated major counter. In the following, we first present the page re-encryption process in existing work [6, 52], which however causes the problem of crash inconsistency for persistent memory. We then present how to guarantee crash consistency during page re-encryption in SecPM.

To re-encrypt a page, all memory lines in this page are read into the last level cache. These memory lines are then re-encrypted one by one using the updated major counter concatenated with a zeroed minor counter, and finally written back into main memory. During re-encrypting these memory lines, a re-encryption status register (RSR) maintained in the memory controller is used to track the re-encryption status of each memory line within a page [52]. The RSR stores the page number and the old major counter of the page. The RSR also maintains a `done` bit for each memory line within the page to indicate whether the corresponding memory line has already been re-encrypted. After all the 64 `done` bits are set to '1' in the RSR, re-encryption of this page is complete and the RSR is freed.

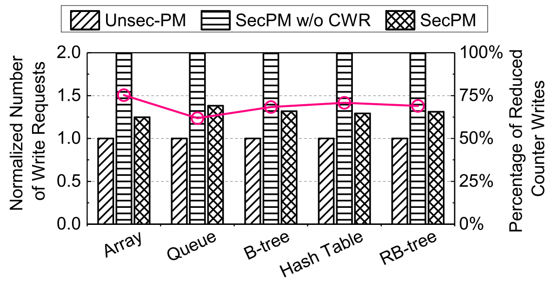
Existing work [52] demonstrates that the latency of page re-encryption can be near-completely reduced from the critical path of processor execution, thus having a negligible impact on the system performance. Since the RSR tracks the re-encryption status of each line within a page that is being re-encrypted, the CPU caches are able to continue to service regular memory access (read and flush) requests to other pages and the processor is not stalled. For an access to a memory line in the page that is being re-encrypted, there are two cases based on the re-encryption status of the line recorded in the RSR. 1) If the line has been already re-encrypted, i.e., its `done` bit is '1', the access normally proceeds. 2) If the line is not re-encrypted, i.e., its `done` bit is '0', the access simply waits for the re-encryption of the line. Moreover, Huang et al. [14] and Swami et al. [43] show that the counter mode encryption can be combined with NVM wear-level techniques [35, 39], and thus the frequency of page re-encryption is significantly reduced.

However, when the page re-encryption process is executed in persistent memory, if a system failure occurs during re-encrypting a page, some memory lines within the page have been re-encrypted but others have not. Moreover, the re-encryption status and page number recorded in the RSR are lost. After recovery, the system does not know which page is being re-encrypted and which memory lines in this page have not been re-encrypted. As a result, the memory lines that have not been re-encrypted cannot be correctly decrypted, thus resulting in an inconsistent state.

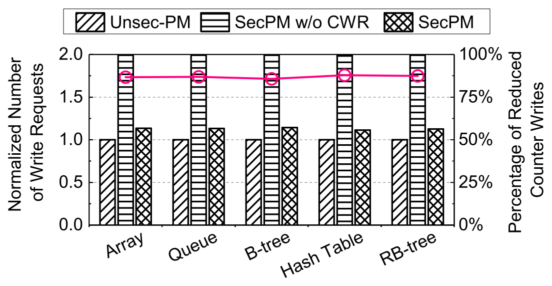
To guarantee crash consistency of page re-encryption, we employ the ADR mechanism (providing battery backup for the write queue) on the RSR. The battery overhead is negligible, since the size of RSR is very small, i.e., 20 bytes, including 32-bit physical page number, 64-bit old major counter and 64-bit `done` bits. The data stored in the RSR are flushed into NVM in case of a system failure using the ADR and loaded into RSR again when the system is recovered. Thus the system knows which page is being re-encrypted and which memory lines in the page have not been re-encrypted, based on the content of the RSR. Hence, the system can continue to complete the page re-encryption after



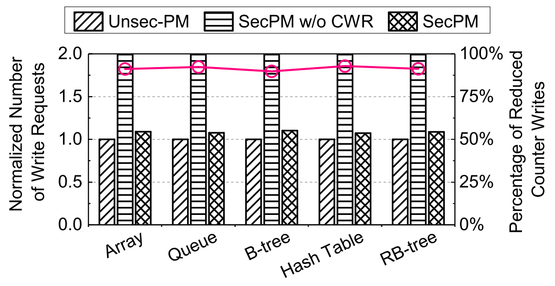
(a) 64B transaction size



(b) 256B transaction size



(c) 1KB transaction size



(d) 4KB transaction size

Figure 11: The numbers of NVM write requests normalized to those of Unsec-PM with different transaction sizes. *(The black line with circles in each figure shows the percentage of reduced counter writes via CWR in SecPM.)*

the recovery from a system failure. Moreover, the process of re-encrypting each memory line is the same as that the memory controller deals with a regular cache line flush as shown in Figure 7. Thus the consistency of writing each re-encrypted line is also ensured by the CWT scheme, and the performance of page re-encryption is also improved by the CWR scheme, since the writes during a page re-encryption have good spatial locality.

4 Performance Evaluation

4.1 Methodology

As real hardware is not available yet for implementing NVMM and the proposed persistence and encryption schemes, we use gem5 [4] with NVMain [33] to evaluate SecPM. NVMain is a cycle-accurate main memory simulator for emerging NVM technologies. The NVM system consists of x86-64 processors running at 2GHz, 32KB L1 data and instruction caches, 512KB L2 caches, and 4MB L3 cache. The counter cache is 1MB. Without loss of generality, we model PCM technologies [8] with 16GB capacity. The PCM latency model is the same as that used in Xu el al.'s work [49]. The encryption and decryption latencies of AES circuit are 40ns [29, 40]. To support the simulation of persistent memory, we employ the `clwb` and `sfence` instructions that have been implemented in the latest gem5.

Since the performance improvement of SecPM mainly comes from the counter write reduction (CWR) scheme as presented in Section 3.4, we compare `SecPM` with the `SecPM w/o CWR` which indicates the SecPM without the proposed CWR scheme. Moreover, we also consider an

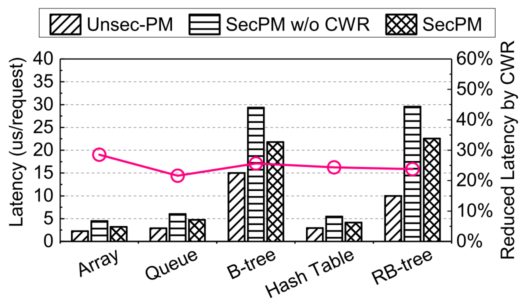
Table 3: The configurations of the NVM system.

Processor	
CPU	4 cores, X86-64 processor, 2 GHz
Private L1 cache	64KB, 8-way, LRU, 2-CPU-cycle latency
Private L2 cache	512KB, 8-way, LRU, 8-CPU-cycle latency
Shared L3 cache	4MB, 8-way, LRU, 30-CPU-cycle latency
Main Memory	
Capacity	16GB, 16 banks in 2 ranks
PCM latency model	tRCD/tCL/tCWD/tFAW/tWTR/tWR =48/15/13/50/7.5/300 ns
En/decryption latency	40 ns
Write queue	32 entries
Counter cache	1 MB, 8-way, LRU, 12-CPU-cycle latency

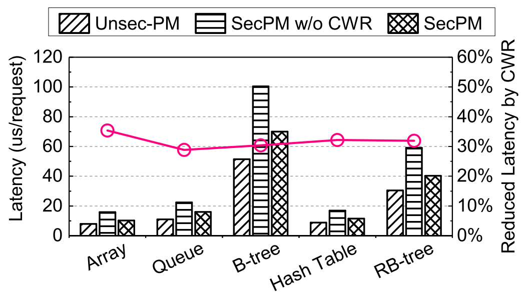
unsecure persistent memory without memory encryption (`Unsec-PM`) as a baseline for comparisons. We do not compare the performance of programmer-transparent SecPM with Liu et al.'s work [29] due to no open-source codes and different design goals.

We use five workloads to evaluate the performance of SecPM as listed in the following. The five workloads are also widely used in existing work on persistent memory [9, 21, 23, 29, 37].

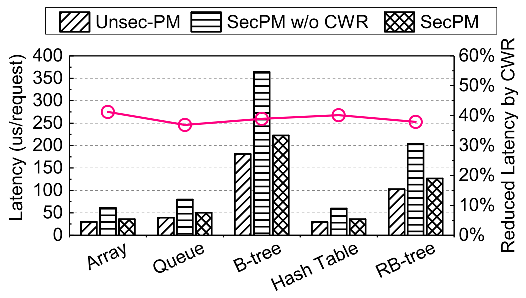
- ***Array.*** Initializing a 1GB array and then randomly swapping entries.
- ***Queue.*** Randomly enqueueing and dequeueing entries in a 1GB queue.
- ***B-tree.*** Inserting random key-value items into a 2GB B-tree based key-value store.



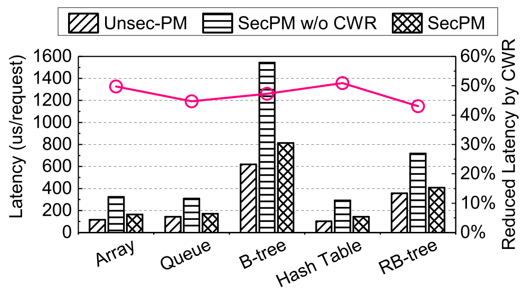
(a) 64B transaction size



(b) 256B transaction size



(c) 1KB transaction size



(d) 4KB transaction size

Figure 12: The average latency of executing each transaction with different transaction sizes. *(The black line with circles in each figure shows the percentage of reduced latency that SecPM uses CWR).*

- ***Hash table.*** Inserting random key-value items into a 2GB hash table based key-value store.
- ***RB-tree.*** Inserting random key-value items into a 2GB red-black tree.

The ACID property (atomicity, consistency, isolation, and durability) of operations in these workloads is ensured via durable transaction, like existing work [28, 29, 31, 37].

4.2 Experimental Results

We first present the experimental results of Unsec-PM, SecPM, and SecPM w/o CWR in terms of the the number of write requests and transaction execution latency. We then evaluate the sensitivity of the experimental results under different configuration parameters.

4.2.1 The Number of Write Requests

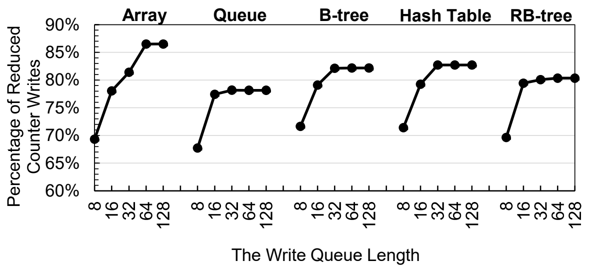
Since the proposed CWR scheme improves the system performance by employing the spatial locality of log and data writes, different transaction request sizes exhibit different spatial locality, hence having a great impact on the system performance. Thus we vary the transaction request sizes in the five workloads from 64B to 4KB to evaluate the number of NVM writes.

Figure 11 shows the number of write requests to NVM normalized to that of Unsec-PM in the five workloads. We observe the SecPM w/o CWR achieves the security and crash consistency but incurs about $2\times$ writes compared with Unsec-PM, whatever the transaction request size is. The reason is that each data write in secure NVM produces two write requests: one for the data and the other for the counter. Compared with the SecPM w/o CWR, SecPM significantly reduces the number of NVM writes. Even in the worst case where the request size is 64B (which means only a cache line is written in a transaction), SecPM reduces $22\% - 37\%$ of counter writes compared with the SecPM w/o CWR, since the transaction data writes have no locality while the log writes have locality. When the transaction request size increases, the locality of data writes significantly increases. SecPM reduces $62\% - 75\%$, $86\% - 88\%$, and $90\% - 93\%$ of counter writes compared with the SecPM w/o CWR, when the transaction sizes are 256B, 1KB, and 4KB, respectively.

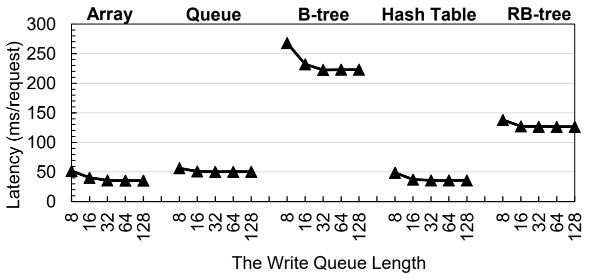
4.2.2 Transaction Execution Latency

We vary the transaction request sizes in the five workloads from 64B to 4KB to evaluate the transaction execution latency.

Figure 12 shows the average latency of executing each transaction with different transaction sizes. We observe that the SecPM w/o CWR increases the transaction execution latency by $2 - 3\times$ across these workloads compared with Unsec-PM, since doubling the number of write requests to NVM significantly degrades the system performance. Compared with the SecPM w/o CWR, SecPM significantly reduces the average transaction execution latency via reducing the number of counter writes. In the worst case where the transaction size is 64B, SecPM still reduces the transaction execution latency by $20\% - 29\%$ compared with

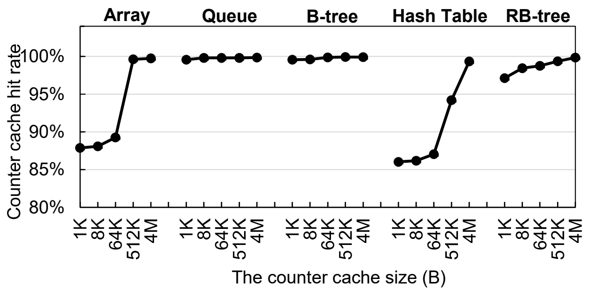


(a) The percentage of reduced counter writes

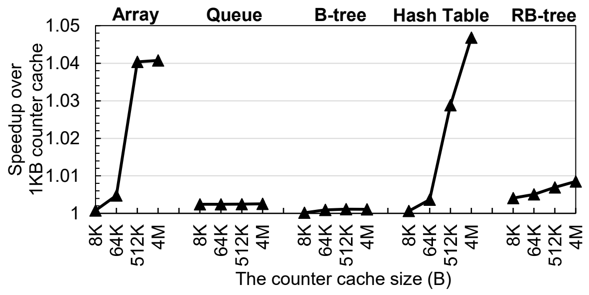


(b) The average latency of executing each transaction

Figure 13: The percentage of reduced counter writes and average latency of executing each transaction with different transaction sizes.



(a) Counter cache hit rate



(b) Speedup over a 1KB counter cache

Figure 14: Evaluating SecPM with different counter cache sizes.

the SecPM w/o CWR, due to reducing 22% – 37% of counter writes as evaluated in Section 4.2.1. When the transaction sizes are 256B, 1KB, and 4KB, SecPM reduces 29% – 36%, 37% – 41%, and 43% – 51% of average execution latency respectively, compared with the SecPM w/o CWR. When the transaction sizes are larger than 1KB, SecPM incurs only a little latency increase, compared with Unsec-PM, which comes from the the latency overhead of data encryption and decryption. In summary, SecPM speeds up the transaction execution by 1.3 × –2.0× via the CWR scheme, due to the reduction in the number of write requests, and achieves the performance close to an unsecure persistent memory system for large transactions.

4.2.3 Sensitivity to Write Queue Size

We use the fixed configurations of 1MB counter cache and 1KB transaction size, and vary the write queue length from 8 to 128 to evaluate the performance in terms of the number of write requests and transaction execution latency.

Figure 13a shows the influence of different write queue lengths on the percentage of reduced counter writes in SecPM, compared with the SecPM w/o CWR. We observe that SecPM reduces more counter writes with longer write queue. The reason is that longer write queue provides more opportunities for the CWR scheme to find and merge more counter writes with the same physical address in the write queue. For most workloads including queue, B-tree, hash table and RB-tree, when the write queue length is larger than 32, the percentage of reduced counter writes increases little. When the write queue length increases from 8 to 128, SecPM reduces 17%, 11%, 11%, 11%, and 10% of more counter writes for array, queue, B-tree, hash table, and RB-tree workloads, respectively.

Figure 13b shows the influence of different write queue lengths on the transaction execution latency in SecPM. We observe that increasing the write queue length decreases the average latency of executing each transaction in all workloads, since longer write queue reduces more counter writes. When the write queue length increases from 8 to 128, SecPM reduces the average transaction execution latency by 31%, 11%, 17%, 27%, and 9% for array, queue, B-tree, hash table, and RB-tree workloads, respectively.

4.2.4 Sensitivity to Counter Cache Size

We use the fixed configurations of 32-entry write queue and 1KB transaction size, and vary the counter cache size from 1KB to 4MB to evaluate the counter cache hit rate and workload execution time.

Figure 14a shows the influence of different counter cache sizes on the cache hit rate in SecPM. We observe that increasing the counter cache size has a great impact on the counter cache hit rates for array, hash table, and RB-tree, but rarely affects those of queue and B-tree. The reason is that the dequeue or enqueue in the queue accesses a continuous memory space and B-tree has the structure that a node continuously stores multiple key-value items, which exhibit good spatial locality for data accesses. In contrast, the random entry swaps in the array, inserting items into random hash locations in the hash table, and the structure

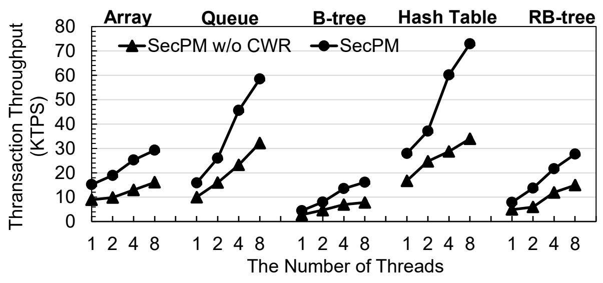


Figure 15: The concurrent transaction throughput in the 1/2/4/8-core systems.

of one item per node in the RB-tree exhibit poor spatial locality for data accesses. The good spatial locality for data accesses produces high counter cache hit rate. When reading a memory line in a page, all counters that decrypt this page are loaded into the counter cache, due to being stored in a memory line. The following accesses to the same page always hit the counter cache. As shown in Figure 14a, when the counter cache size increases from 1KB to 4MB, the cache hit rate is improved by 12%, 14%, and 3% for array, hash table and rb-tree workloads.

Figure 14b shows the influence of different counter cache sizes on the overall execution time of workloads in SecPM. The execution time of workloads under different counter cache sizes are normalized to those under 1KB counter cache size. We observe that different counter cache sizes have little impact on the execution time of queue and B-tree workloads. For array, hash table, and RB-tree, the execution performance is improved respectively by 4%, 5%, and 1%, when the counter cache size increases from 1KB to 4MB.

4.2.5 Multiple-core Performance

We evaluate the performance of SecPM in a multi-core system, where each thread executes the same transactions on different cores. We use the configurations of 32-entry write queue, 1KB transaction size, and 1MB counter cache. Figure 15 shows the transaction throughput of different workloads in the 1/2/4/8-core systems. We observe that the transaction throughput of workloads in SecPM scales well with the increase of the number of cores. Moreover, compared with the SecPM w/o CWR, SecPM improves the transaction throughput by 64%, 81%, 90% and 95% on average in a 1/2/4/8-core system respectively.

5 Related Work

• ***Secure Non-volatile Memory.*** As NVM suffers from the data remanence vulnerability, data security in NVM has been widely studied. DEUCE [55] proposes a dual-counter encryption scheme to reduce the write traffic in the encrypted NVM by re-encrypting only the modified words in a memory line. Based on DEUCE, SECRET [44] further avoids the re-encryption of zero-content words in a memory line to reduce bit writes. Silent Shredder [3] reduces NVM writes in the encrpted NVM by eliminating the full-zero cache line writes produced from data shredding. DeWrite [59] proposes a lightweight deduplication scheme to enhance the performance and endurance of the encrypted NVM via eliminating duplicate-content writes. All these schemes on the encrypted NVM mainly focus on reducing the writes of encrypted data to NVM, which do not consider the crash consistency in the secure NVM. Moreover, some existing works focusing on memory authentication in NVM [36, 54], which are orthogonal to our work, as discussed in Section 3.1.

• ***Crash Consistency in Non-volatile Memory.*** To achieve data persistence, various durable transaction systems, such as NV-Heaps [9], Mnemosyne [47], DudeTM [28], NVML [20], and DCT [22], are proposed to manage persistent data with crash consistency in NVM. Moreover, multiple NVM-based file systems, such as BPFS [10], PMFS [13], Mojim [56], NOVA [50], and NOVA-Fortis [51], are proposed to achieve the improvement of storage performance by leveraging the byte-addressable benefit of NVM, which also provide the crash consistency guarantee by employing copy-based techniques, e.g., logging, copy-on-write (shadowing page), and replication. All these schemes are built on the un-encrypted NVM, without considering the memory encryption on NVM.

SecPM aims to ensure both the security and crash consistency of NVM. In terms of addressing the crash consistency of secure NVM, the most related work comes from Liu et al. [29], which is the first work to efficiently ensure that data and its counter are atomically persisted via the selective counter-atomicity. For a different design goal, i.e., programmer transparency, SecPM performs only slight modifications on the memory controller to achieve crash consistency. Moreover, SecPM achieves significant performance improvement and write reduction via a counter write reduction scheme.

6 Conclusion

This paper proposes SecPM to achieve both the security and persistence in non-volatile main memory. SecPM leverages a counter cache write-through scheme with a register to guarantee crash consistency of durable transaction as well as atomic writes in secure NVM. Moreover, a counter write reduction (CWR) scheme is introduced to improve the system performance. These schemes are implemented with slight modifications only on the hardware layer, which are transparent for programmers and applications. Thus programs and applications running on an un-encrypted NVM can be directly executed on a secure NVM with SecPM. Experimental results demonstrate that SecPM achieves the performance close to an unsecure persistent memory system for large transactions.

References

[1] Akinaga, H., and Shima, H. Resistive random access memory (ReRAM) based on metal oxides. *Proceedings of the IEEE 98*, 12 (2010), 2237–2251.

[2] Apalkov, D., Khvalkovskiy, A., Watts, S., Nikitin, V., Tang, X., Lottis, D., Moon, K., Luo, X., Chen, E., Ong, A., et al. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC) 9*, 2 (2013), 13:1–13:35.

[3] Awad, A., Manadhata, P., Haber, S., Solihin, Y., and Horne, W. Silent Shredder: Zero-cost shredding for secure non-volatile main memory controllers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[4] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News 39*, 2 (2011), 1–7.

[5] Chen, S., and Jin, Q. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment 8*, 7 (2015), 786–797.

[6] Chhabra, S., Rogers, B., Solihin, Y., and Prvulovic, M. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing (ICS)* (2011).

[7] Chhabra, S., and Solihin, Y. i-NVMM: a secure non-volatile main memory system with incremental encryption. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2011).

[8] Choi, Y., Song, I., Park, M.-H., Chung, H., Chang, S., Cho, B., Kim, J., Oh, Y., Kwon, D., Sunwoo, J., et al. A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)* (2012).

[9] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[10] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)* (2009).

[11] Daemen, J., and Rijmen, V. *The design of Rijndael: AES-the advanced encryption standard.* Springer Science & Business Media, 2013.

[12] David, K., Jeremy, P., and Tom, W. AMD memory encryption. *AMD White Paper* (2016).

[13] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (Eurosys)* (2014).

[14] Huang, F., Feng, D., Hua, Y., and Zhou, W. A wear-leveling-aware counter mode for data encryption in non-volatile memories. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)* (2017).

[15] Hudson, R. L., Saha, B., Adl-Tabatabai, A.-R., and Hertzberg, B. C. Mcrt-malloc: A scalable transactional memory allocator. In *Proceedings of the 5th International Symposium on Memory Management (ISMM)* (2006).

[16] Hwang, D., Kim, W.-H., Won, Y., and Nam, B. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)* (2018).

[17] Intel Corporation. Introducing Intel Optane Technology - Bringing 3D XPoint Memory to Storage and Memory Products. https://goo.gl/xRFooQ, 2015.

[18] Intel Corporation. Deprecating the PCOMMIT Instruction. https://goo.gl/k5CsyA, 2016.

[19] Intel Corporation. Intel® Architecture Instruction Set Extensions and Future Features Programming Reference. https://goo.gl/y629n9, 2017.

[20] Intel Corporation. Persistent memory programming. http://pmem.io/, 2018.

[21] Kolli, A., Gogte, V., Saidi, A., Diestelhorst, S., Chen, P. M., Narayanasamy, S., and Wenisch, T. F. Language-level persistency. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017).

[22] Kolli, A., Pelley, S., Saidi, A., Chen, P. M., and Wenisch, T. F. High-performance transactions

for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[23] Kolli, A., Rosen, J., Diestelhorst, S., Saidi, A., Pelley, S., Liu, S., Chen, P. M., and Wenisch, T. F. Delegated persist ordering. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016).

[24] Lee, H., Chen, Y., Chen, P., Gu, P., Hsu, Y., Wang, S., Liu, W., Tsai, C., Sheu, S., Chiang, P., et al. Evidence and solution of over-reset problem for hfo x based resistive memory with sub-ns switching speed and high endurance. In *Proceedings of the 2010 IEEE International Electron Devices Meeting (IEDM)* (2010).

[25] Lee, M.-J., Lee, C. B., Lee, D., Lee, S. R., Chang, M., Hur, J. H., Kim, Y.-B., Kim, C.-J., Seo, D. H., Seo, S., Chung, U.-I., Yoo, I.-K., and Kim, K. A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta2o5-x/tao2-x bilayer structures. *Nature materials 10*, 8 (2011), 625–630.

[26] Lee, S. K., Lim, K. H., Song, H., Nam, B., and Noh, S. H. WORT: write optimal radix tree for persistent memory storage systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2017).

[27] Lipmaa, B. H., Rogaway, P., and Wagner, D. Ctr-mode encryption, comments to nist concerning aes modes of operations. In *NIST Workshop on Modes of Operation* (2000).

[28] Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W., and Ren, J. 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).

[29] Liu, S., Kolli, A., Ren, J., and Khan, S. Crash consistency in encrypted non-volatile main memory systems. In *Proceedings of the IEEE 24th International Symposium on High-Performance Computer Architecture (HPCA)* (2018).

[30] Mueller, W., Aichmayr, G., Bergner, W., Erben, E., Hecht, T., Kapteyn, C., Kersch, A., Kudelka, S., Lau, F., Luetzen, J., et al. Challenges for the dram cell scaling to 40nm. In *IEEE International Electron Devices Meeting (IEDM)* (2005).

[31] Nalli, S., Haria, S., Hill, M. D., Swift, M. M., Volos, H., and Keeton, K. 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).

[32] Pelley, S., Chen, P. M., and Wenisch, T. F. Memory persistency. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2014).

[33] Poremba, M., Zhang, T., and Xie, Y. Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems. *IEEE Computer Architecture Letters 14*, 2 (2015), 140–143.

[34] Qureshi, M. K., Franceschini, M. M., and Lastras-Montano, L. A. Improving read performance of phase change memories via write cancellation and write pausing. In *Proceedings of the IEEE 16th International Symposium on High-Performance Computer Architecture (HPCA)* (2010).

[35] Qureshi, M. K., Karidis, J., Franceschini, M., Srinivasan, V., Lastras, L., and Abali, B. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2009).

[36] Rakshit, J., and Mohanram, K. ASSURE: authentication scheme for secure energy efficient non-volatile memories. In *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2017).

[37] Ren, J., Zhao, J., Khan, S., Choi, J., Wu, Y., and Mutlu, O. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)* (2015).

[38] Seo, J., Kim, W.-H., Baek, W., Nam, B., and Noh, S. H. Failure-atomic slotted paging for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[39] Seong, N. H., Woo, D. H., and Lee, H.-H. S. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)* (2010).

[40] Shi, W., Lee, H.-H. S., Ghosh, M., Lu, C., and Boldyreva, A. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)* (2005).

[41] Shin, S., Tirukkovalluri, S. K., Tuck, J., and Solihin, Y. 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).

[42] Suh, G. E., Clarke, D., Gassend, B., Dijk, M. v., and Devadas, S. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2003).

[43] Swami, S., and Mohanram, K. ACME: advanced counter mode encryption for secure non-volatile memories. In *Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC)* (2018).

[44] Swami, S., Rakshit, J., and Mohanram, K. SECRET: smartly encrypted energy efficient non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)* (2016).

[45] Thoziyoor, S., Ahn, J. H., Monchiero, M., Brockman, J. B., and Jouppi, N. P. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *International Symposium on Computer Architecture (ISCA)* (2008).

[46] Volos, H., Magalhaes, G., Cherkasova, L., and Li, J. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference (Middleware)* (2015).

[47] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[48] Wong, H.-S. P., Raoux, S., Kim, S., Liang, J., Reifenberg, J. P., Rajendran, B., Asheghi, M., and Goodson, K. E. Phase change memory. *Proceedings of the IEEE 98*, 12 (2010), 2201–2227.

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

[50] Xu, J., and Swanson, S. 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).

[51] Xu, J., Zhang, L., Memaripour, A., Gangadharaiah, A., Borase, A., Da Silva, T. B., Swanson, S., and Rudoff, A. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)* (2017).

[52] Yan, C., Englender, D., Prvulovic, M., Rogers, B., and Solihin, Y. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2006).

[53] Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K. L., and He, B. NV-Tree: reducing consistency cost for nvm-based single level systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2015).

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

[55] Young, V., Nair, P. J., and Qureshi, M. K. DEUCE: Write-efficient encryption for non-volatile memories. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[56] Zhang, Y., Yang, J., Memaripour, A., and Swanson, S. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[57] Zhou, P., Zhao, B., Yang, J., and Zhang, Y. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2009).

[58] Zuo, P., Hua, Y., and Wu, J. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2018).

[59] ZUO, P., HUA, Y., ZHAO, M., ZHOU, W., AND GUO, Y. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2018).