SuperMem: Enabling Application-transparent Secure Persistent Memory with Low Overheads

Pengfei Zuo
Huazhong University of Science and Technology, China
University of California Santa Barbara, USA
pfzuo@hust.edu.cn

Yu Hua
Huazhong University of Science and Technology, China
(Corresponding Author)
csyhua@hust.edu.cn

Yuan Xie
University of California Santa Barbara, USA
yuanxie@ece.ucsb.edu

ABSTRACT

Non-volatile memory (NVM) suffers from security vulnerability to physical access based attacks due to non-volatility. To ensure data security in NVM, counter mode encryption is often used by considering its high security level and low decryption latency. However, the counter mode encryption incurs new persistence problem for crash consistency guarantee due to the requirement for atomically persisting both data and its counter. To address this problem, existing work requires a large battery backup or complex modifications on both hardware and software layers due to employing a write-back counter cache. The large battery backup is expensive and software-layer modifications limit the portability of applications from the un-encrypted NVM to the encrypted one. Our paper proposes SuperMem, an application-transparent secure persistent memory by leveraging a write-through counter cache to guarantee the atomicity of data and counter writes without the needs of a large battery backup and software-layer modifications. To reduce the performance overhead of a baseline write-through counter cache, SuperMem leverages a locality-aware counter write coalescing scheme to reduce the number of write requests by exploiting the spatial locality of counter storage and data writes. Moreover, SuperMem leverages a cross-bank counter storage scheme to efficiently distribute data and counter writes to different banks, thus speeding up writes by exploiting bank parallelism. Experimental results demonstrate that SuperMem improves the performance by about 2× compared with an encrypted NVM with a baseline write-through counter cache, and achieves the performance comparable to an ideal secure NVM that exhibits the optimal performance of an encrypted NVM.

MICRO '52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358290

CCS CONCEPTS

• **Hardware** → **Non-volatile memory**; • **Security and privacy** → **Hardware attacks and countermeasures**.

KEYWORDS

Non-volatile memory, memory encryption, crash consistency

ACM Reference Format:
Pengfei Zuo, Yu Hua, and Yuan Xie. 2019. SuperMem: Enabling Application-transparent Secure Persistent Memory with Low Overheads. In *MICRO '52: The 52nd Annual IEEE/ACM International Symposium on Microarchitecture, October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358290

1 INTRODUCTION

As DRAM suffers from limited scalability and high power leakage [31, 41], non-volatile memories (NVM), such as PCM [43], ReRAM [1], STT-RAM [2], and 3D XPoint [18], 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.

First, 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 non-trivial [32, 42], a factor 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 [27, 48]. Hence, cache line flush, memory barrier, and log-based mechanisms are used to ensure the crash consistency [29, 45].

Second, the non-volatility of NVM also causes the security problem of data remanence vulnerability [3, 50], since NVM still retains data after systems are powered down. In the legacy DRAM-based main memory, when using encryption to protect data, the encrypted data are stored in disks, while raw data are retained in main memory [14]. 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 easily stream out the data from the DIMM. Hence, memory encryption becomes important to ensure data security in NVM. Counter mode encryption [28, 50] is usually used in the secure NVM, due to its low decryption latency and high security level. Counter mode encryption uses a secret key associated with the line address and a counter to generate a one-time pad (OTP) via the AES encryption engine. In the counter mode encryption, we encrypt the data via XORing the plaintext with OTP, and decrypt the data by XORing the ciphertext data with OTP. The counter mode encryption generates OTP via the counter buffered in an on-chip counter cache, while in the meantime, the data is fetched from memory. The parallel execution thus hides the AES computation latency efficiently.

Nevertheless, it is challenging to ensure both crash consistency and data security in NVM. This is because each data write to the encrypted NVM generates two write requests, i.e., one for the data and the other for its counter. To guarantee crash consistency, the two writes have to be persisted at the same time. If a system failure occurs when the data is persisted into NVM but its counter is not, the data fails to be decrypted upon the system recovery due to no correct counter. Similarly, if a system failure occurs when the counter is persisted into NVM but its data is not, the old-version data in NVM fails to be correctly decrypted. Unfortunately, current computer systems cannot atomically perform the two write requests to NVM, since the data is evicted from CPU caches and the counter is evicted from the counter cache managed by the memory controller. Programmers can actively flush the data in CPU caches by using existing cache line flush and memory barrier instructions for data persistence. However, these instructions cannot control counter writes from the counter cache.

To guarantee crash consistency of the encrypted NVM and reduce its performance overhead, some existing works [3, 56] use *a write-back counter cache* with battery backup. These schemes persist counters in the counter cache into NVM by using the battery power on system failures. However, in practice, the counter cache is usually hundreds of kilobytes or even megabytes [3, 30, 47, 56] and thus the battery backup for supporting the large counter cache is expensive and occupies large chip areas [49]. Modern processor vendors only provide a small battery backup for the Asynchronous DRAM Refresh (ADR) [20, 30, 37] with the small persistent domain of tens of entries in the write queue of the memory controller. Without using the large battery backup, existing work [30] proposes a selective counter atomicity (SCA) scheme with *a write-back counter cache* by adding new primitives in the programming language including `CounterAtomic` variable and `counter_cache_writeback()` function. These primitives enable programmers to explicitly flush specified counters from the write-back counter cache into NVM. Moreover, the SCA scheme adds a counter write queue into the memory controller, which enables the data and its counter to wait for each other. The counter atomicity is achieved by performing modifications on both software and hardware layers including programming language, compiler, and memory controller. As a result, the applications initially running on a system with the un-encrypted NVM cannot directly run on a system with the encrypted NVM.

Our paper proposes **SuperMem** (pronunciation similar to "Superman"), an application-transparent **S**ec**u**re **per**sistent **Mem**ory without a large battery backup and software-layer modifications. SuperMem employs *a write-through counter cache* that enables crash consistency guarantee of the encrypted NVM to be much easier compared with using a write-back counter cache. However, the baseline write-through counter cache incurs more counter write requests, thus decreasing the system performance. SuperMem further improves the system performance from two aspects, i.e., reducing the number of counter write requests and speeding up memory writes. Specifically, SuperMem leverages a locality-aware counter write coalescing (CWC) scheme to reduce counter write requests by exploring and exploiting the spatial locality of counter storage and data writes. Moreover, SuperMem leverages a cross-bank counter storage (XBank) scheme to efficiently distribute data and counter writes to different banks, thus speeding up writes by exploiting bank parallelism. In summary, SuperMem shows how to use the simple yet effective CWC and XBank schemes to enable an encrypted NVM with a write-through counter cache to deliver a high performance comparable to an ideal secure NVM that exhibits the optimal performance of an encrypted NVM. This paper makes the following contributions:

- ***Counter write reduction by exploiting data locality.*** We propose a locality-aware counter write coalescing (CWC) scheme to improve system performance in SuperMem via significantly reducing the number of NVM write requests from counters. By leveraging the spatial locality of counter storage and data writes, i.e., the counters of the data with successive physical addresses are stored in the same memory line, the CWC scheme merges different counter writes to the same memory line in the write queue.
- ***Write speedup by leveraging bank parallelism.*** We propose a cross-bank counter storage (XBank) scheme to efficiently distribute data and counter writes to different banks in SuperMem. Writes to different banks can be handled by memory in parallel. Thus the XBank scheme speeds up writes by leveraging bank parallelism.
- ***Application-transparent implementation.*** The implementation of SuperMem 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 applications initially running on an un-encrypted NVM can be directly executed on an encrypted NVM with SuperMem.
- ***Experimental evaluation.*** We have implemented and evaluated the SuperMem in gem5 [6] with NVMain [33]. Experimental results show the CWC scheme reduces up to 50% of write requests in the encrypted NVM with a write-through counter cache, and the XBank scheme improves the system performance by up to 2×.

Thus SuperMem achieves the performance comparable to an ideal secure NVM that presents the optimal performance of an encrypted NVM.

2 BACKGROUND AND MOTIVATION

In this section, we first present the background of data persistence and security issues in NVM. We then present the gap between the persistence and security guarantees.

2.1 Consistency Guarantee for Persistence

In order to correctly persist data into NVM, it is important to guarantee data crash consistency. However, modern CPUs and memory controllers usually reorder memory writes, which usually results in data inconsistency in case of system failures. Thus cache line flush and memory barrier instructions are used to enforce write ordering [23, 29, 32]. The cache line 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 in the write queue into NVM but was deprecated later by Intel [19], due to the use of asynchronous DRAM refresh (ADR) mechanism [20, 30, 37]. The 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.

Moreover, the size of atomic memory write in modern computer systems is 8 bytes, which is equal to the memory bus width for 64-bit CPUs. If data larger than 8 bytes is being updated and a system failure occurs before completing the update, the data will be corrupted. Copy-based mechanisms [23, 29] such as logging and copy-on-write are used to avoid the partial update. For example, the logging technique first writes the new data (redo logging) or old data (undo logging) into a log, and then updates the data in place. If a system failure occurs during the updating process, the data can be recovered based on the log. Nevertheless, copy-based mechanisms are expensive due to writing more data. Some crafted data structures, e.g., wB$^+$-tree [7], WORT [27], FAST&FAIR [17], and level hashing [55], are proposed to exploit the atomic write of NVM to ensure data crash consistency, thus avoiding the overhead of copy-based mechanisms. For example, a bit map associated with the data is used to indicate which data are valid or invalid and the bit map is smaller than an atomic write. The data structure changes the bit map to atomically switch the data from the old version to the new one.

2.2 Memory Encryption for Security

2.2.1 Threat Model. Our threat model is similar to existing work on secure NVM [3, 30, 40, 50, 56], which aims to protect NVM from two well-known physical access based attacks, including stolen DIMM and bus snooping attacks[1]. 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. To defend against these attacks, memory encryption is important and non-trivial.

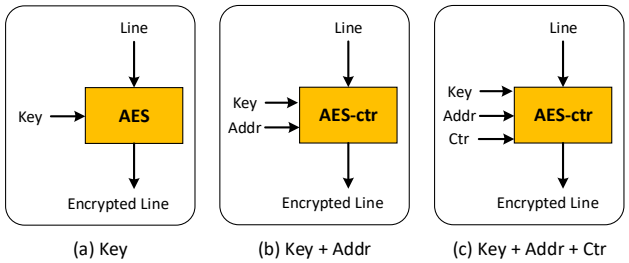


Figure 1: The encryption methods (a) using a global key; (b) using a key and line address; (c) using a key, line address and a counter (ctr).

2.2.2 Security Guarantee via Encryption. A straightforward method to encrypt a memory line is to use a block cipher algorithm, e.g., AES [13], with a global key, as shown in Figure 1a. 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, 50]. Using the key along with the line address to encrypt each line is more secure, as shown in Figure 1b, 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.

A secure method is to encrypt each memory line by using the global key and line address in conjunction with a per-line counter, as shown in Figure 1c. The counter increases by one on each write and hence consecutive memory writes to the same line are encrypted with different keys, achieving high security level.

2.2.3 Latency Reduction via One Time Pad. As memory reads are in the critical path of program execution, memory encryption causes high latency of decryption that follows each memory read due to serial execution, as shown in Figure 2a. The serial execution significantly degrades the system performance. Unlike it, counter mode encryption [28] is able to reduce the decryption latency from the critical path of memory reads via the one-time pad (OTP) technique, and hence has been widely used in encrypted memory systems [3, 9, 40, 50, 56]. 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

[1] We do not consider bus tampering attacks in the threat model like existing work [3, 30, 40, 50, 56]. These attacks can be defended via Merkle Trees based authentication techniques [38], which are orthogonal to our work.

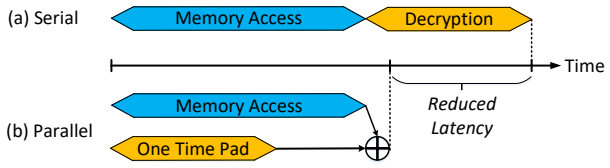


Figure 2: Reducing the decryption latency using One Time Pad (OTP).

decryption latency in the memory access latency, as shown in Figure 2b.

2.2.4 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 [28, 40, 50, 56]. 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 3. 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. To reduce the generation time of OTPs, the memory controller manages an on-chip counter cache to buffer the recently-accessed counters [30, 40, 56].

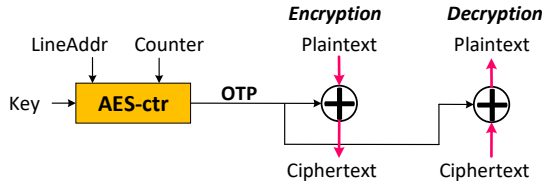


Figure 3: The encryption and decryption processes in counter mode encryption.

2.3 The Gap between Persistence and Security

Each data write to the encrypted NVM produces two write requests: data and its counter. To guarantee crash consistency, the two writes have to be persisted at the same time. As shown in Figure 4a, if a system failure occurs after the counter has been persisted into NVM but before its data is persisted, the data in NVM cannot be correctly decrypted upon the system recovery since there is no correct counter. On the other hand, as shown in Figure 4b, if a system failure occurs after the data has been persisted into NVM but before its counter is persisted, the data still cannot be decrypted. When both data and its counter are persisted in an atomic manner, crash consistency is guaranteed as shown in Figure 4c. However, current computer systems fail to atomically perform the two write requests to NVM because the data is evicted from CPU caches and the counter is evicted from the counter cache managed by the memory controller. Programmers are able to actively flush the data in CPU caches by using existing cache line flush and memory barrier instructions. However, these instructions cannot control counter writes in the counter cache [30, 39, 54].

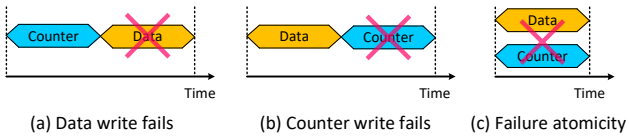


Figure 4: Different cases when system failures occur.

Even though some copy-based mechanisms have been used in NVM to guarantee crash consistency, they do not work in the encrypted NVM. We analyze the effectiveness of the durable transaction [23, 29] that is a commonly-used solution for crash consistency guarantee by using logging. When a system failure occurs in the different stages of a durable transaction executed in an encrypted NVM, the recoverability of the durable transaction is 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 it is unknown whether the counters encrypting the log are correctly persisted, since the cache line flush and memory barrier instructions fail to handle 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 reason, 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.

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

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**

To address the inconsistency problem in the encrypted NVM, existing work [30] proposed the concept of the selective counter-atomicity (SCA), which indicates that either both data and its associated counter have been simultaneously persisted or not. In the write-back counter cache, the counter is written into NVM only when being evicted. Therefore, SCA adds the new primitive `counter_cache_writeback()` function in the programming language to enable programmers to explicitly flush specified counter cache lines into NVM. Moreover, SCA adds a counter write queue into the memory controller, which enables the data and its counter to wait for each other. However, the data and counters stalled in the write queues decrease the system performance. The authors

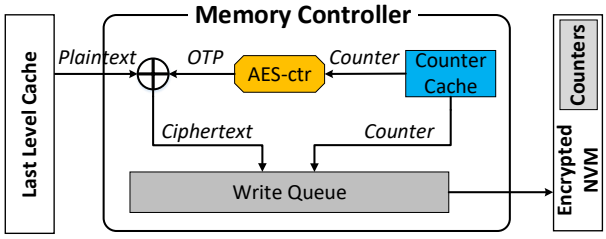


Figure 5: The hardware architecture of SuperMem. ***(The ADR is applied to the write queue and hence cache lines reaching the write queue are considered durable.)***

observe some writes do not immediately affect the recoverability of data in a consistent state. For example, writes in the mutate stage of a durable transaction are not consistent when a system failure occurs. The transaction can still be recovered by using the logging. Therefore, SCA selectively relaxes the atomicity of these writes to improve performance and adds the new primitive `CounterAtomic` to enable the memory controller to identify the data writes that cannot relax atomicity. SCA is effective when using copy-based mechanisms to guarantee crash consistency in the encrypted NVM but requires the use of new programming primitives. When porting the applications initially running on a system with an un-encrypted NVM to the system with an encrypted NVM, programmers have to modify the program codes of applications by adding the new primitives in the right places.

3 THE SUPERMEM DESIGN

3.1 An Architectural Overview

Our paper proposes SuperMem, an application-transparent solution to guarantee crash consistency of the encrypted NVM. SuperMem employs a write-through counter cache with a register (§3.2) that enables crash consistency to be much easier than existing work using a write-back counter cache. When the data is written into NVM, its corresponding counter is also written into NVM following the data. Thus SuperMem is application-transparent which does not require programmers to actively flush counters from the counter cache into NVM. However, using a write-through counter cache always generates two write requests for each data write, decreasing the system performance. We propose an efficient cross-bank counter storage (Xbank) scheme (§3.3) to distribute the data write and its counter write to different banks, improving the system performance by leveraging band parallelism. Moreover, we also propose a counter write coalescing (CWC) scheme (§3.4) in SuperMem to significantly reduce the number of write requests by leveraging the spatial locality of counter and data writes.

The hardware architecture of SuperMem 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 with the ADR mechanism presented

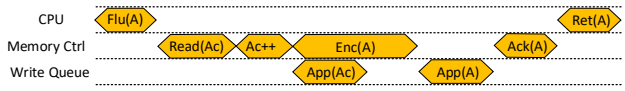


Figure 6: The sequence that the memory controller deals with a cache line flush by using a write-through counter cache. ***(Flu(A): flushing the cache line A into NVM; Ac: the counter of A; App(Ac): appending Ac in the write queue; Enc(A): encrypting A; Ret(A): the flush of A is retired.)***

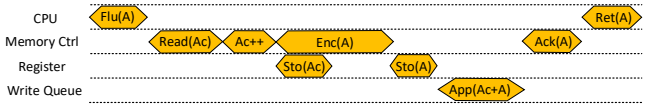


Figure 7: The sequence that the memory controller deals with a cache line flush by using a write-through counter cache with a register. ***(Sto(Ac): storing Ac in the register.)***

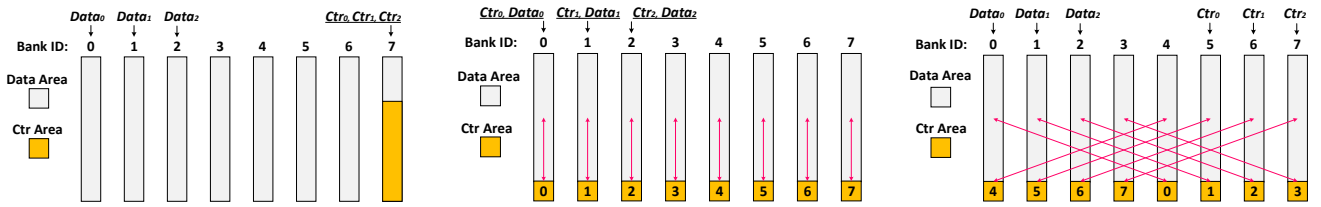
in Section 2.1. The write-through scheme is performed in the counter cache. The Xbank scheme is used by the memory controller. The CWC scheme is performed in the write queue to merge counter writes. As a whole, SuperMem only performs slight hardware modifications on the memory controller, which are transparent for programmers and applications. Thus programs and applications running on an un-encrypted NVM can be directly executed on an encrypted NVM with SuperMem.

3.2 Write-through Counter Cache

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

Figure 6 shows the sequence diagram that the memory controller deals with a cache line flush or a naturally evicted CPU cache line in SuperMem. 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 write-through 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 write-through scheme. Hence, if the data is persisted, its counter has been also persisted.

However, simply performing the baseline write-through scheme cannot ensure crash consistency when the counter has been persisted but the data has not. Specifically, we consider



(a) SingleBank storage **(b) SameBank storage** **(c) XBank storage**

Figure 8: Different counter storage approaches. ***(The number on a counter (Ctr) area indicates the bank number storing its corresponding data, e.g., in Figure 8c, the Ctr area with the number 4 in Bank 0 stores the counters of data in Bank 4.)***

the flush for cache line A shown in Figure 6 as an example. If a system failure occurs after appending the counter of A (i.e., *App(Ac)*) before appending A (i.e., *App(A)*) in the write queue, the counter of A (i.e., A_c) is updated and persisted in NVM using the ADR mechanism [20, 30, 37] but A is not. After the system is recovered from the failure, the old value of A cannot be decrypted using the new counter.

To address this problem, we add a register for the AES encryption engine. 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 encrypting the data, we first store the encrypted data in the register ($Sto(A)$), and then simultaneously append the encrypted data and its counter in the write queue ($App(Ac + A)$). We observe that either data and its associated counter simultaneously exist in the write queue or not, by using a register. As a result, the crash consistency of data and its counter is ensured in SuperMem. 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).Reads and writes in such a small register are very fast. The use of the register has a negligible performance overhead.

3.3 Cross-bank Counter Storage

In existing secure NVM work with counter mode encryption [3, 30, 50, 56], counters are generally stored in a continuous area in NVM (SingleBank), as shown in Figure 8a. The storage approach is efficient for the write-back counter cache, since most counter writes are buffered in the counter cache and aren't written into NVM. However, when employing a write-through counter cache, each data write produces a counter write to the counter storage bank. Thus the bank storing counters becomes a bottleneck, decreasing the system performance. For example, three data write requests, i.e., $Data_0$, $Data_1$, and $Data_2$, are sent to Banks 0, 1, and 2, respectively, as shown in Figure 8a. All their counter write requests, i.e., Ctr_0, Ctr_1, and Ctr_2, are sent to the counter storage bank, i.e., Bank 7. The three data write requests can be served simultaneously due to the parallelism among banks. However, the three counter write requests have to be served one by one, thus blocking the following counter write requests and degrading the system performance.

To avoid the performance bottleneck from accessing the single bank of counters, one straightforward solution is to store the counters of data into their local banks (SameBank), as shown in Figure 8b. Each data write and its counter write are sent to the same bank. In this case, however, the average processing time that each bank serves for a data write is doubled, compared with an un-encrypted NVM, since each bank needs to serially serve for two write requests for each data write. For example, Bank 0 needs to serially serve for $Data_0$ and Ctr_0, as shown in Figure 8b.

To reduce the processing time of write requests, we propose *a cross-bank counter storage (XBank) scheme* which stores each data and its counter into different banks instead of the same bank. The two banks storing the data and their corresponding counters are one-to-one way as shown in Figure 8c. Moreover, the interval between the two bank numbers storing data and their corresponding counters should be as large as possible. It is because the operating system usually allocates continuous memory space for the same application which may locate in the adjacent banks. In this case, the data and counter writes from the same application are easily sent to the same bank, causing bank access conflicts. Therefore, in the XBank scheme, if the data is stored in Bank X, we store its corresponding counter in Bank $(X + N/2)\%N$, where N is the total number of banks, e.g., N is 8 in the example shown in Figure 8c. By performing the XBank scheme, data and counter writes are distributed into different banks, efficiently reducing bank access conflicts. Hence, writes are sped up by leveraging bank parallelism. For example, as shown in Figure 8c, the three data writes, i.e., $Data_0$, $Data_1$, and $Data_2$, and their corresponding counter writes, i.e., Ctr_0, Ctr_1, and Ctr_2, can be served at the same time.

In general cases that only partial banks are accessed at the same time, the XBank scheme becomes efficient due to the strength of parallel execution. In the high-load cases that almost all banks are simultaneously accessed, the XBank scheme may have the similar performance to SameBank due to high competition of bank accesses. However, it is still much better than SingleBank. Even though in the worst case that all memory banks are accessed at the same time, SuperMem still delivers high performance as evaluated in Section 5.1.2, due to efficiently synergizing the XBank scheme and the CWC scheme presented in the next subsection.

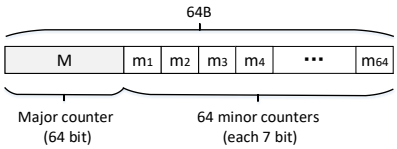


Figure 9: The counter storage of 64 lines within a physical page.

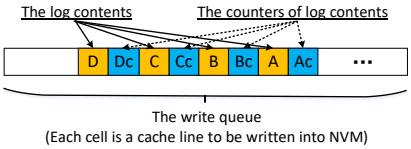


Figure 10: An illustration of the write queue when writing a log.

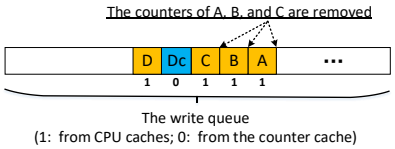


Figure 11: The write queue when writing a log via the CWC scheme.

3.4 Locality-aware Counter Write Coalescing

In the encrypted 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. We propose *a locality-aware counter write coalescing (CWC) scheme* in SuperMem to reduce the number of counter writes via exploiting the spatial locality of counter storage, log and data writes.

3.4.1 Spatial Locality of Counter Storage. SuperMem exploits the split counter mode encryption as shown in Figure 9, since it not only reduces the storage overhead of counters [47] but also facilitates our proposed counter write coalescing to reduce counter writes. The counter mode encryption uses a shared major counter (M) for a page and 64 minor counters (m_1, m_2, ..., m_{64}) each for a memory line in the 4KB page, as shown in Figure 9. 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 [47]. The process of re-encrypting a page is presented in Section 3.4.4. The 64-bit major counter cannot 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., $10^7 - 10^9$ for PCM [16, 34, 52] and $10^8 - 10^{12}$ for ReRAM [25, 26].

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 exhibit spatial locality, since programs usually allocate a contiguous memory region for a transaction. Hence, the cache lines flushed into the contiguous region have contiguous physical addresses. For example, a transaction inserts a 1KB key-value item into a key-value store maintained in NVM, which flushes 16 cache lines with contiguous physical addresses.

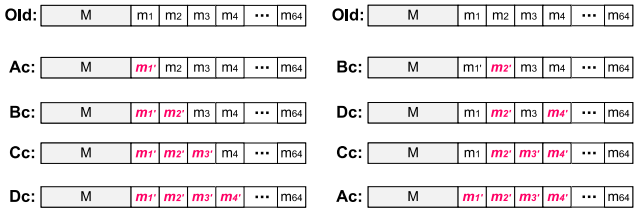


Figure 12: The contents of the corresponding counter cache lines when the CPU cache lines *A*, *B*, *C* and *D* are flushed in order or out of order.

3.4.3 Counter Write Coalescing Scheme. Based on the locality existing in counter storage, log and data writes, Figure 10 shows the write queue during flushing the log entry of a transaction. 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 9, 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.

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 12a. We observe that the counter cache line A_c only contains the updated minor counter m_1', B_c contains m_1' and m_2', and C_c contains m_1', m_2', and m_3'.

Moreover, the 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. 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 12b.

According to the above observations and insights, we present *a counter write coalescing (CWC) 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 merge these cache lines with the same physical address. Moreover, instead of merging the latter cache line into the former cache line in the write queue, we directly remove the former cache line which is advantageous to delay the counter cache line write for merging more writes.

For example, as shown in Figure 10, we directly remove A_c, B_c, and C_c instead of merging D_c into them. The deletions of the former counter cache lines do not cause any loss of data, since the new counter cache line contains the updated contents of the removed ones as shown in Figure 12. 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 to facilitate fast identification of counter cache lines. The flag is used to distinguish whether a cache line is from CPU caches or the counter cache, i.e., '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 CWC scheme, the new write queue is shown in Figure 11. 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 CWC scheme. By using the CWC scheme, only $64 + 1 = 65$ cache lines are written into NVM, thus reducing almost half of NVM writes.

3.4.4 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 [8, 47], which may cause the problem of crash inconsistency for persistent memory. We then present how to perform slight modifications to guarantee crash consistency during page re-encryption.

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 [47]. 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 64 `done` bits are set to '1' in the RSR, re-encryption in this page is complete and the RSR is freed.

However, if a system failure occurs during re-encrypting a page in persistent memory, some memory lines within the page have been re-encrypted but others have not. In this case, 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 fail to be correctly decrypted, thus resulting in an inconsistent state.

To guarantee crash consistency of page re-encryption, SuperMem employs 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 contents of the RSR. Hence, the system continues to complete the page re-encryption after 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. The consistency of writing each re-encrypted line is also ensured by the write-through scheme, and the performance of page re-encryption is also improved by the CWC and XBank schemes.

Table 2: Configurations of the simulated system.

Processor	
CPU	8 cores, X86-64 processor, 2 GHz
Private L1 cache	64KB, 8-way, LRU, 2-cycle latency
Private L2 cache	512KB, 8-way, LRU, 15-cycle latency
Shared L3 cache	4MB, 8-way, LRU, 30-cycle latency
Main Memory	
Capacity	8GB, 8 banks
PCM latency model	tRCD/tCL/tCWD/tFAW/tWTR/tWR =48/15/13/50/7.5/300 ns
Write queue	32 entries
Counter cache	256KB, 8-way, LRU, 8-cycle latency

4 EVALUATION METHODOLOGY

As real hardware is not available yet for implementing the proposed persistence and encryption schemes, we use gem5 [6] with NVMain [33] to evaluate SuperMem. 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 shared L3 cache. The counter cache is 256KB. Without loss of generality, we model PCM technologies [10] with 8GB capacity. The PCM latency model is the same as that used in Xu et al.'s work [44]. We model an AES pipeline encryption engine with the 24-cycle encryption latency, like prior work [4, 49]. To support the simulation of persistent memory, we employ the `clwb` and `sfence` instructions that have been implemented in the latest gem5. We compare the proposed SuperMem with the following schemes.

- *An un-encrypted NVM (**Unsec**).* It is a baseline NVM system without using memory encryption.
- *An ideal secure NVM with a write-back counter cache (**WB**).* It uses an ideal write-back counter cache in which only the evicted dirty counter cache lines are written into NVM. We assume the ideal write-back counter cache is battery-backup and thus has no any counter-atomicity performance overhead. Therefore, the ideal secure NVM, i.e., the WB scheme in our experiments, presents the optimal performance of an encrypted NVM. SCA [30] uses a write-back counter

cache without using the large battery backup, but the counter-atomicity data and counters waiting for each other in write queues incur performance overhead as discussed in Section 2.3. Therefore, SCA has lower performance than the WB scheme.

- *A write-through scheme (**WT**).* It uses a baseline write-through counter cache that flushes both data and its counter into NVM for each data write. The WT scheme does not require any software-layer modifications to achieve counter crash consistency.
- *A write-through scheme with CWC (**WT+CWC**).* It uses a write-through counter cache and employs our proposed counter write coalescing (CWC) scheme presented in Section 3.4 to reduce counter writes.
- *A write-through scheme with XBank (**WT+XBank**).* It uses a write-through counter cache and employs our proposed cross-bank counter storage (XBank) scheme presented in Section 3.3 to speed up memory write requests.

We use five workloads to evaluate the performance of SuperMem in the following. The ACID property (atomicity, consistency, isolation, and durability) of operations in these workloads is ensured via durable transaction. The five workloads are widely used in existing work on persistent memory [11, 22, 24, 30, 36].

- **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 1GB B-tree based key-value store.
- **Hash Table.** Inserting random key-value items into a 1GB hash table based key-value store.
- **RB-tree.** Inserting random key-value items into a 1GB red-black tree.

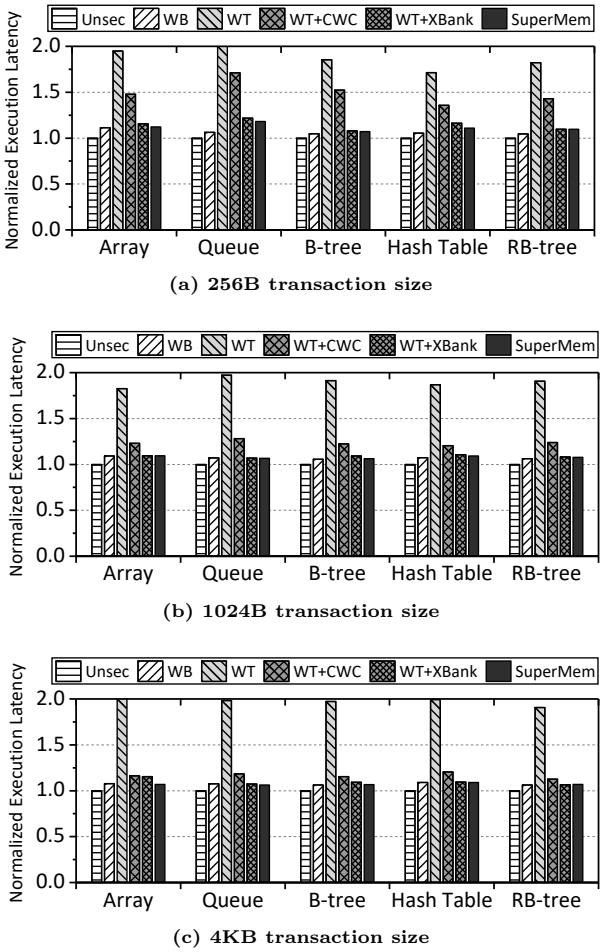


Figure 13: The average latency of executing transaction requests with different transaction sizes normalized to those of an un-encrypted NVM.

5 PERFORMANCE EVALUATION

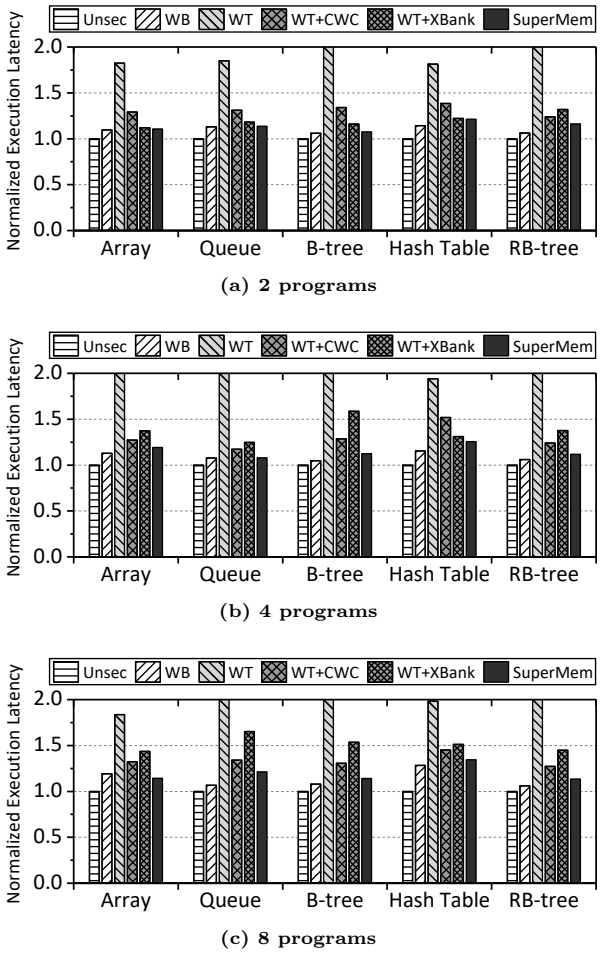
In this section, we first investigate the impacts of different schemes on transaction execution latency and the number of write requests. We then evaluate the sensitivity of experimental results under different configuration parameters.

5.1 Transaction Execution Latency

5.1.1 Single-core Performance. Figure 13 shows the average latency of executing each transaction in different workloads. We observe that compared with the Unsec scheme (i.e., without secure guarantee) , the ideal WB scheme slightly increases the transaction execution latency due to only increasing a few counter memory writes as shown in Section 5.2. The WT scheme increases the transaction execution latency by about $1.7\times - 2\times$ across these workloads compared with the Unsec scheme, since doubling the number of write requests to NVM significantly degrades the system performance.

Our proposed SuperMem is based on the WT scheme. In order to examine the efficiency of counter write coalescing (CWC) and cross-bank counter storage (Xbank) in SuperMem, we evaluate the performance of the WT+CWC and WT+Xbank schemes, respectively. The size of transaction requests impacts the data storage locality and thus the performance of the CWC scheme. Hence, we respectively evaluate the performance of the transactions with a small 256B request size (including only 4 cache-line writes), a 1024B request size, and a 4096B request size.

Compared with the WT scheme, WT+CWC reduces the average transaction execution latency by 17%−24% even with a very small transaction request size, as shown in Figure 13a. With increasing the transaction request size to 1024B and 4096B, WT+CWC reduces execution latency by 30% − 35% and 40% − 48%, as shown in Figures 13b and 13c. This is because the transaction with larger size has better spatial locality, and thus more counter writes are merged by the CWC scheme. Compared with the WT scheme, WT+XBank reduces the transaction execution latency by up to 45% among all workloads. This is due to the fact that the XBank scheme efficiently distributes the data write and its counter



(a) 2 programs

(b) 4 programs

(c) 8 programs

Figure 14: The average latency of executing transaction requests normalized to those of an un-encrypted NVM in multi-core applications.

write into different banks, overlapping the write latency of the two writes. By employing both CWC and Xbank schemes, SuperMem improves about 2× performance compared with the WT scheme and achieves the approximate performance as the ideal WB scheme. The transaction execution latency of SuperMem is slightly higher than the Unsec scheme, due to encryption and decryption overheads.

5.1.2 Multi-core Performance. We evaluate the performance of SuperMem in a multi-core system, where each thread executes the same program on different cores. We use the configurations of 32-entry write queue, 1KB transaction size, and 256KB counter cache. We enable each program to have the memory footprint size that is equal to the size of a memory bank. Thus every program must access at least one bank. For the 8-program experiments, the 8 memory banks are used at the same time, which presents the worst-case performance for the XBank scheme as described in Section 3.3. The experimental results are shown in Figure 14.

We observe that compared with Unsec, the WT scheme increases the transaction execution latency by $1.8\times - 2.4\times$, like its single-core performance. Both the WT+CWC and WT+XBank schemes significantly reduce the transaction execution latency compared with the WT scheme. However, different from the single-core performance, the WT+CWC scheme has a lower transaction execution latency than the WT+XBank scheme for most workloads in the 4-program and 8-program experiments. It means that the CWC scheme obtains more performance benefits than the XBank scheme in multi-core applications due to more bank access competitions. By employing both CWC and XBank schemes, SuperMem also achieves the approximate multi-core performance as the WB scheme.

5.2 The Number of Write Requests

Figure 15 shows the number of write requests to NVM normalized to that of the Unsec scheme in the five workloads. Since the proposed CWC scheme improves the system performance by employing the spatial locality of log and data writes, different transaction request sizes exhibit different spatial localities, thus impacting the number of memory writes. Therefore, we vary the transaction request sizes in the five workloads from 256B to 4096B to evaluate the number of NVM writes.

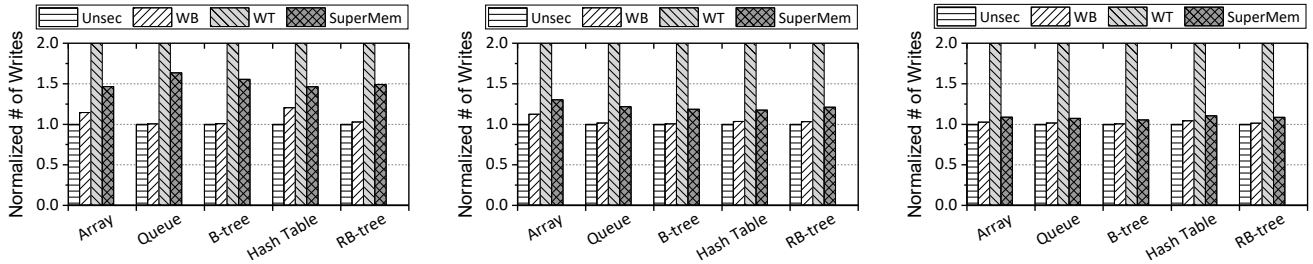
We observe the WB scheme increases the number of memory writes by $3\% - 16\%$ due to evicting the dirty counter cache lines from the counter cache into NVM for the small 256B transaction request size. With the increase of the transaction request size, the number of memory writes in the WB scheme is reduced, since counter accesses have better spatial locality, thus achieving higher counter cache hit ratio for the WB scheme. The WT scheme incurs 2× memory writes compared with the Unsec scheme whatever the transaction request size is. The reason is that each data write in the secure NVM produces two write requests: one for the data and the other for the counter.

Compared with the WT scheme, SuperMem significantly reduces the number of NVM writes. Even in the cases with a small transaction request size as shown in Figure 15a, SuperMem reduces $20\% - 27\%$ of memory writes, since the transaction data writes have low locality while the log writes always have good locality. When the transaction request size increases, the locality of data writes significantly increases. SuperMem reduces memory writes by $35\% - 42\%$, and $45\% - 48\%$ compared with the WT scheme, when the transaction sizes are 1024B and 4096KB, respectively.

5.3 Sensitivity to Write Queue Size

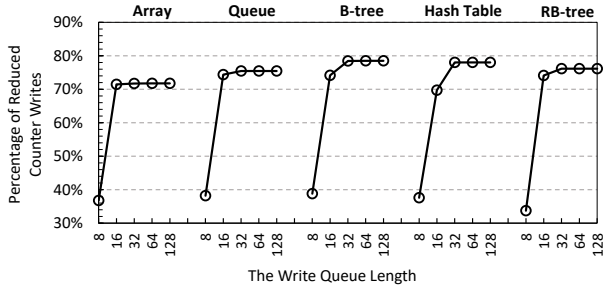
We use the fixed configurations of 256KB counter cache and 1KB transaction size and vary the write queue length from 8 to 128 to evaluate the performance in terms of the number of write requests and transaction execution latency.

Figure 16a shows the influence of different write queue lengths on the percentage of reduced counter writes in SuperMem, compared with the WT scheme. We observe that SuperMem reduces more counter writes with longer write queue. The reason is that longer write queue provides more

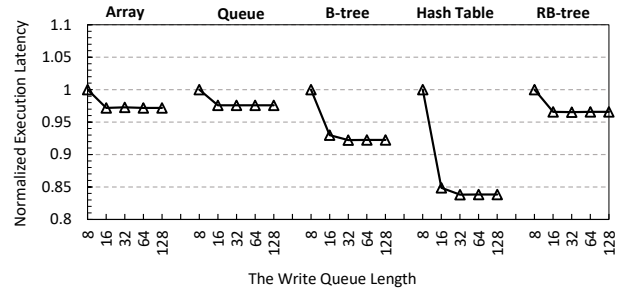


(a) 256B transaction size **(b) 1KB transaction size** **(c) 4KB transaction size**

Figure 15: The numbers of NVM write requests normalized to those of Unsec with different transaction sizes.

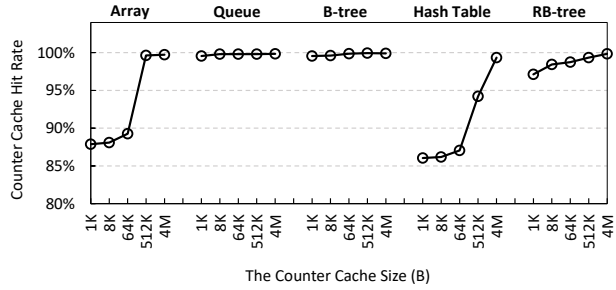


(a) The percentage of reduced counter writes

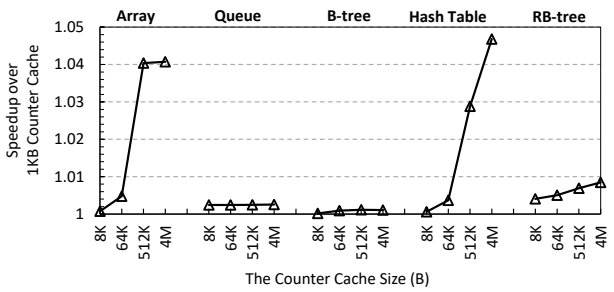


(b) Execution latency normalized to an 8-entry write queue

Figure 16: The percentage of reduced counter writes and average execution latency with different write queue lengths.



(a) Counter cache hit rate



(b) Speedup over a 1KB counter cache

Figure 17: The counter cache hit rates and the performance of SuperMem with different counter cache sizes.

opportunities for the CWC scheme to find and merge more counter writes with the same physical address in the write queue. When the write queue length increases from 8 to 128, SuperMem reduces 35%, 37%, 40%, 40%, and 42% of more counter writes for array, queue, B-tree, hash table, and RB-tree workloads, respectively. When the write queue length is larger than 32, the percentage of reduced counter writes increases little for most workloads. Hence, a write queue with the length of 32 is enough and reasonable for the CWC.

Figure 16b shows the influence of different write queue lengths on the transaction execution latency in SuperMem. 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 32, the average transaction execution latency is reduced by 3%, 3%, 8%, 15%, and 4% for array, queue, B-tree, hash table, and RB-tree workloads, respectively.

5.4 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 17a shows the influence of different counter cache sizes on the cache hit rate in SuperMem. We observe that increasing the counter cache size has a substantial 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 access continuous

memory space. In the B-tree structure, a node continuously stores multiple key-value items which exhibit good spatial locality for data accesses. In contrast, random entry swaps in the array, item insertions into random hash locations in the hash table, and the structure of one item per node in the RB-tree exhibits 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 17a, 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 17b shows the influence of different counter cache sizes on the overall execution time of workloads in SuperMem. 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.

6 RELATED WORK

Secure NVM. As NVM suffers from the data remanence vulnerability, data security in NVM has been widely studied. DEUCE [50] 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 [40] 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 encrypted NVM by eliminating the full-zero cache line writes produced from data shredding. DeWrite [56] 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 aim to reduce the writes of encrypted data to NVM, which do not focus on crash consistency in the secure NVM. Moreover, some existing works focusing on memory authentication in NVM, such as ASSURE [35], Triad-NVM [5], and Anubis [53], are orthogonal to our work, as discussed in Section 2.2.1.

Crash Consistency in NVM. To achieve data persistence, various durable transaction systems, such as Mnemosyne [42], NV-Heaps [11], DudeTM [29], NVML [21], and DCT [23], are proposed to manage persistent data with crash consistency guarantee in NVM. Moreover, multiple NVM-based file systems, such as BPFS [12], PMFS [15], Mojim [51], NOVA [45], and NOVA-Fortis [46], 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 memory encryption on NVM.

Crash Consistency in Secure NVM. In order to offer crash consistency of the encrypted NVM, existing schemes [3, 56] consider to use a write-back counter cache with the aid of battery backup power. However, due to the high costs and limited chip areas available [49], the backup battery becomes difficult to be practical in real implementations. Modern processor vendors only provide a small battery backup for the Asynchronous DRAM Refresh (ADR) mechanism [20, 30, 37], and thus a small-size domain—i.e., tens of entries in the write queue—can be persistent. Without using the large battery backup, Liu et al. [30] propose the selective counter-atomicity (SCA) scheme with a write-back counter cache, which ensures that data and its counter are atomically persisted with low overheads by using new programming primitives. Unlike these existing works that use the write-back counter cache but need a large battery backup or software-layer modifications, SuperMem shows how to use a write-through counter cache to guarantee crash consistency of the encrypted NVM while having low performance overheads via the simple yet efficient XBank and CWC schemes. Moreover, Ye et al. [49] propose Osiris to relax the counter persistence during application execution and recover wrong counters after a system failure by leveraging error-correction codes and Merkle Tree. However, Osiris incurs long counter recovery time when the system is recovered from a failure and the recovery time linearly increases with the memory size [53]. In contrast, our proposed SuperMem and SCA [30] do not need to recover counters due to strict counter persistence during application execution.

7 CONCLUSION

This paper proposes SuperMem to achieve both security and persistence in non-volatile main memory. SuperMem leverages a write-through counter cache scheme with a register to guarantee crash consistency in the encrypted NVM. Moreover, a counter write coalescing scheme is introduced to reduce the number of write requests and a cross-bank counter storage scheme is employed to reduce the processing time of write requests. 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 an encrypted NVM with SuperMem. Experimental results show that SuperMem achieves the performance comparable to an ideal secure NVM exhibiting the optimal performance of an encrypted NVM.

ACKNOWLEDGMENTS

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202, and National Natural Science Foundation of China (NSFC) under Grant 61772212. This work was also supported in part by NSF 1816833, 1730309, and CRISP, one of six centers in JUMP, a SRC program sponsored by DARPA. Pengfei Zuo was also supported by a grant from the China Scholarship Council (CSC).

REFERENCES

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.

[2] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. 2013. 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] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. 2016. 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)*.

[4] A. Awad, Y. Wang, D. Shands, and Y. Solihin. 2017. ObfusMem: A low-overhead access obfuscation for trusted memories. In *Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 107–119.

[5] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. 2019. Triad-NVM: Persistency for Integrity-Protected and Encrypted Non-Volatile Memories. In *Proceedings of the 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

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

[8] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. 2011. SecureME: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing (ICS)*.

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

[10] Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, et al. 2012. A 20nm 1.8 V 8Gb PRAM with 40MB/s program bandwidth. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*.

[11] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. 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)*.

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

[13] Joan Daemen and Vincent Rijmen. 2013. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media.

[14] Kaplan David, Powell Jeremy, and Woller Tom. 2016. AMD Memory Encryption. *AMD White Paper* (2016).

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

[16] Yuncheng Guo, Yu Hua, and Pengfei Zuo. 2018. A Latency-optimized and Energy-efficient Write Scheme in NVM-based Main Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[17] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*.

[18] Intel Corporation. 2015. Introducing Intel Optane Technology - Bringing 3D XPoint Memory to Storage and Memory Products. https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/.

[19] Intel Corporation. 2016. Deprecating the PCOMMIT Instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[20] Intel Corporation. 2017. Intel® Architecture Instruction Set Extensions and Future Features Programming Reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

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

[22] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. 2017. Language-level persistency. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*.

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

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

[25] HY Lee, YS Chen, PS Chen, PY Gu, YY Hsu, SM Wang, WH Liu, CH Tsai, SS Sheu, PC Chiang, et al. 2010. 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)*.

[26] Myoung-Jae Lee, Chang Bum Lee, Dongsoo Lee, Seung Ryul Lee, Man Chang, Ji Hyun Hur, Young-Bae Kim, Chang-Jung Kim, David H Seo, Sunae Seo, U-In Chung, In-Kyeong Yoo, and Kinam Kim. 2011. 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.

[27] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)*.

[28] By H Lipmaa, P. Rogaway, and D. Wagner. 2000. CTR-Mode Encryption, Comments to NIST concerning AES Modes of Operations. In *NIST Workshop on Modes of Operation*.

[29] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. 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)*.

[30] Sihang Liu, Aasheesh Kolli, Jinglei Ren, and Samira Khan. 2018. Crash Consistency in Encrypted Non-Volatile Main Memory Systems. In *Proceedings of the IEEE 24th International Symposium on High-Performance Computer Architecture (HPCA)*.

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

[32] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2014. Memory persistency. In *the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*.

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

[34] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. 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)*.

[35] Joydeep Rakshit and Kartik Mohanram. 2017. ASSURE: Authentication Scheme for SecURE energy efficient non-volatile memories. In *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[36] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*.

[37] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. 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)*.

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

[39] Shivam Swami and Kartik Mohanram. 2018. ACME: Advanced counter mode encryption for secure non-volatile memories. In *Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*.

[40] Shivam Swami, Joydeep Rakshit, and Kartik Mohanram. 2016. SECRET: smartly EnCRypted energy efficient non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)*.

[41] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B Brockman, and Norman P Jouppi. 2008. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *International Symposium on Computer Architecture (ISCA)*.

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

[43] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

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

[45] Jian Xu and Steven Swanson. 2016. 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)*.

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

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

[48] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)*.

[49] Mao Ye, Clayton Hughes, and Amro Awad. 2018. 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)*.

[50] Vinson Young, Prashant J Nair, and Moinuddin K Qureshi. 2015. 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)*.

[51] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. 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)*.

[52] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*.

[53] Kazi Abu Zubair and Amro Awad. 2019. Anubis: ultra-low overhead and recovery time for secure non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*.

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

[55] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. 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)*.

[56] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. 2018. 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)*.