Improving the Performance and Endurance of Encrypted Non-volatile Main Memory through Deduplicating Writes

Pengfei Zuo, Yu Hua, Ming Zhao[†], Wen Zhou, Yuncheng Guo
Wuhan National Laboratory for Optoelectronics
School of Computer, Huazhong University of Science and Technology, China
[†]Arizona State University, USA
Corresponding author: Yu Hua (csyhua@hust.edu.cn)

***Abstract*—Non-volatile memory (NVM) technologies are considered as promising candidates of the next-generation main memory. However, the non-volatility of NVMs leads to new security vulnerabilities. For example, it is not difficult to access sensitive data stored on stolen NVMs. Memory encryption can be employed to mitigate the security vulnerabilities, but it increases the number of bits written to NVMs due to the diffusion property and thereby aggravates the NVM wear-out induced by writes. To address these security and endurance challenges, this paper proposes DeWrite, a secure and deduplication-aware scheme to enhance the performance and endurance of encrypted NVMs based on a new in-line deduplication technique and the synergistic integrations of deduplication and memory encryption. Specifically, it performs low-latency in-line deduplication to exploit the abundant cache-line-level duplications leveraging the intrinsic read/write asymmetry of NVMs and light-weight hashing. It also opportunistically parallelizes the operations of deduplication and encryption and allows them to co-locate the metadata for high time and space efficiency. DeWrite was implemented on the gem5 with NVMain and evaluated using 20 applications from SPEC CPU2006 and PARSEC. Extensive experimental results demonstrate that DeWrite reduces on average 54% writes to encrypted NVMs, and speeds up memory writes and reads of encrypted NVMs by $4.2\times$ and $3.1\times$, respectively. Meanwhile, DeWrite improves the system IPC by 82% and reduces 40% of energy consumption on average.**

I. Introduction

As DRAM technology suffers from high power leakage and limited scalability, non-volatile memory technologies (NVMs), such as PCM, ReRAM, and STT-RAM, have been proposed as promising candidates of the next-generation main memory [1]–[4], due to having the advantages of high density, high scalability, and requiring near-zero standby power [5], [6]. However, NVMs also face the following obstacles in order to be effectively used in memory systems.

The first problem is the limited write endurance and performance. NVMs typically have limited write endurance, e.g., $10^7 - 10^8$ writes for PCM [6]–[9]. Writes on NVMs not only consume the limited endurance, but also cause higher latency (i.e., $3 - 8\times$) and energy overhead than reads [10], [11]. Moreover, NVMs are expected to store data as persistent memory for instantaneous failure recovery [12]–[14]. In persistent memory, writes are on the critical path of application execution, since achieving data consistency needs to ensure the ordering of memory writes [15]–[19]. Thus a processor has to stall and wait for a memory write to be completed before issuing the next one. Hence, reducing write operations is non-trivial in NVMs.

The second problem is data remanence vulnerability, i.e., NVMs still retain data after systems are powered down due to their non-volatility. In general, when encryption is used to protect the privacy of data, the encrypted data are retained in disks, while raw data are maintained in main memory [20]. If a DRAM DIMM is stolen, data are quickly lost due to the volatility without information leakage. In contrast, if an NVM DIMM is physically removed from the system, an attacker can directly stream out the data from the DIMM [21]–[24]. Hence, memory encryption becomes important to enhance the data security in NVMs. Unfortunately, memory encryption in NVMs exacerbates the problem of write endurance, as a good encryption algorithm generally has the strong diffusion property, which means that the change of a single bit in the plaintext leads to the change of about half of the bits in the ciphertext [25].

The diffusion property makes existing *subline-level* (i.e., bit-level) write reduction techniques, such as Data Comparison Write (DCW) [9] and Flip-N-Write (FNW) [26], ineffective for encrypted non-volatile main memory (NVMM) [21]–[23]. These techniques reduce the number of bits written to NVMM based on the observation that only a small number of bits are modified for a write. DCW only writes the modified bits in a cache line to NVMM by comparing old and new data. FNW further inverts the data if more than 50% of the bits are modified, which ensures that the number of bits written to NVMM is no more than half of bits in a cache line. With the diffusion property of encryption, the change of a single bit in a cache line will cause half of the bits to be modified when written to the encrypted NVMM [22], [25]. Hence, existing techniques like DCW and FNW cannot achieve significant data reduction for encrypted NVMM, which remains to be an important and challenging problem.

There is, however, abundant data duplication at the *line level* which can be exploited to reduce the number of writes to

secure NVMM. Specifically, we observe that a large number of cache lines written to the memory are identical to existing ones in the memory in many real-world applications. From the results of our experiments, the duplicate lines written to memory account for 58% on average in the 20 applications from SPEC CPU2006 [27] and PARSEC [28], and reach upwards 98% in some applications. These observations motivate us to perform cache-line-level deduplication on secure NVMM. Eliminating a large number of duplicate writes not only extends the endurance of NVMM, but also significantly improves the system performance. First, eliminating duplicate writes efficiently removes the high write latency off from the critical path of application execution. In the meantime, eliminating duplicate writes also speeds up read and non-duplicate write requests by reducing their waiting time. When a write request is served by an NVM bank, the following read/write requests to the same bank are blocked and wait until the current write request is completed [29]. If many duplicate write requests are avoided, the waiting time of the following read/write requests is significantly reduced, thus improving the read/write performance. However, a number of challenges need to be addressed when designing a secure NVMM that leverages deduplication to reduce writes.

The first challenge is how to perform in-line deduplication in NVMM without the decrease of system performance. In order to reduce writes to NVMM, deduplication needs to be performed in-line, i.e., duplicate cache lines have to be identified and eliminated before being written to NVMM. Traditional memory deduplication techniques [30]–[33] perform out-of-line deduplication, which cannot reduce writes to memory because duplicate data are first written into main memory and then identified and merged in the background. Furthermore, since main memory is highly latency-sensitive, duplicate write detection is not allowed to incur high latency as it is performed in the critical path of memory writes. Traditional in-line deduplication solutions [34]–[37] use cryptographic hash functions, e.g., SHA-1 [38] and MD5 [39], to compute the fingerprints of data for detecting duplication. Data are considered to be duplicate if their fingerprints are matched. However, computing the cryptographic hash is expensive and not suitable for in-line deduplication of NVMM. For example, the latency of computing MD5/SHA-1 in hardware implementations is generally more than 300 ns [40], [41], which is close to the latency of an NVM write.

The second challenge is how to integrate deduplication with NVM encryption while delivering good performance. Duplication detection determines whether a cache line is duplicate to existing data in NVMM. If a cache line is duplicate, the write is cancelled. Otherwise, it is encrypted and written to NVMM. Thus duplication detection and data encryption are executed serially in the critical path of memory writes, which incur high latency to the writes of non-duplicate cache lines. Moreover, both encryption and deduplication have heavy metadata overhead. To ensure data security, counter mode encryption is often employed to encrypt data in NVMM due to its low decryption overhead [21]–[23], [41], [42], which requires metadata such as the per-line counters. At the same time, to eliminate duplicate lines, deduplication also requires metadata including the per-line hashes and address mappings. The storage and processing of these metadata incur substantial time and space overheads.

To address these challenges, we propose DeWrite, a novel solution for enhancing both the lifetime and performance of secure NVMM with new in-line deduplication technique and synergistic integration schemes of deduplication and encryption. Specifically, DeWrite makes the following contributions:

- ***Light-weight Deduplication Leveraging Asymmetric Reads and Writes.*** To perform low-latency in-line deduplication, DeWrite proposes a light-weight deduplication scheme for NVMM. DeWrite computes the light-weight hash of a cache line. If the hash of the cache line matches that of an existing line in NVMM, DeWrite reads the line and compares the corresponding data to confirm duplication. DeWrite thus eliminates a duplicate write at the cost of a read latency, improving the system performance due to the intrinsic read/write asymmetry of NVMs where write latency is much higher than read latency (i.e., $3 \sim 8\times$) [10], [29].
- ***Efficient Synergization of Deduplication and Encryption via Parallelism and Metadata Colocation.*** To achieve good performance in deduplication and NVM encryption, DeWrite opportunistically performs the two operations in parallel based on a simple yet effective duplication prediction scheme. If a cache line is predicted to be non-duplicate, DeWrite detects duplication in parallel with data encryption to reduce write latency. Otherwise, DeWrite detects duplication without encrypting data to save the computation (and energy) overhead from encryption. Moreover, DeWrite proposes a co-located metadata storage scheme between deduplication and encryption, by embedding the per-line counters into the null locations in the data structures for supporting deduplication, thus reducing the space overhead from counter storage.
- ***System Implementation and Evaluation.*** We have implemented DeWrite on the gem5 [43] with NVMain [44], and comprehensively evaluated its performance using SPEC CPU2006 [27] and PARSEC [28]. Experimental results show DeWrite eliminates on average 54% of writes to secure NVMM. Furthermore, DeWrite speeds up the memory writes and reads of secure NVMM by $4.2\times$ and $3.1\times$, respectively, and reduces the energy overhead by 40% on average, while incurring only 6.25% metadata storage overhead.

II. Background and Motivation

A. Threat Models

Like existing threat models [21]–[23], our paper aims to protect NVMM from two well-known physical access based attacks, including stolen DIMM and bus snooping attacks. Specifically, due to not the user of a computer, an attacker fails to visit or control the software systems, but can physically access to the NVMM via theft, repair, or improper disposal. In the stolen DIMM attack, due to the non-volatility of NVMs, an attacker stealing the NVM DIMM or acting as a machine repairman can directly stream out all the data from the DIMM.

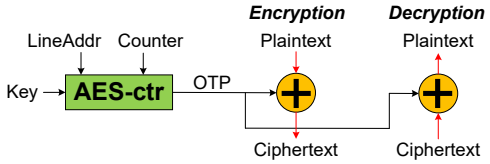


Fig. 1: Counter mode encryption.

In the bus snooping attack, since NVMM is accessed through the memory bus, an attacker can insert a bus snooper or a memory scanner in the bus to obtain the data communicated between the processor chip and NVMM.

B. Memory Encryption

To defend against the stolen DIMM attack, the data in the NVMM should be encrypted. To further defend against the bus snooping attack, we should encrypt the data in the CPU side instead of the memory side, thus avoiding the plaintext passing through the memory bus [21]–[23]. In general, there are two CPU-side memory encryption models, including direct encryption and counter mode encryption, to encrypt the data in the memory. First, in the *direct encryption*, each cache line is encrypted by employing a block cipher algorithm, e.g., AES [45], when written back to the memory from the last level cache, and decrypted after being read from the memory. However, direct encryption incurs high decryption latency in the critical path of memory reads [21], [22], thus decreasing the system performance. Second, in the *counter mode encryption*, data decryption can be executed in parallel with memory read, thus reducing the decryption latency. Counter mode encryption generates a one-time pad (OTP) using a counter and encrypts/decrypts data by XORing the plaintext/ciphertext with the OTP, as shown in Figure 1. Counters are buffered in an on-chip counter cache managed by the memory controller. For a data access, if its counter is found in the counter cache, the OTP is computed in parallel with the memory read, thus hiding the decryption latency in memory access latency. Only the slight latency of XOR operation is added into the critical path of memory read.

The security of counter mode encryption is ensured if each OTP is never reused for data encryption [41], [46], [47]. 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 1. Thus data stored at different addresses are encrypted by different OTPs. Moreover, the per-line counter increases on each write and generates different OTPs for data rewrites of the same address. Hence, the OTPs are never reused.

C. Observation and Motivation

The DeWrite scheme proposed in this paper leverages the counter mode encryption to achieve low decryption latency [21]–[23], [41], [42], and more importantly, it addresses the lack of consideration for write endurance in memory encryption. The use of encryption in NVMs in fact exacerbates the write endurance due to the strong diffusion property [25], which renders the existing bit-level write reduction techniques

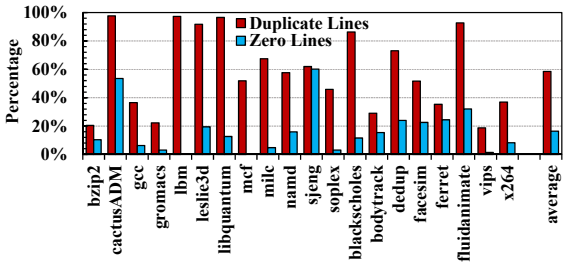


Fig. 2: The percentage of duplicate lines. *(The first 12 applications are from the SPEC CPU2006 and the following 8 applications are from the PARSEC.)*

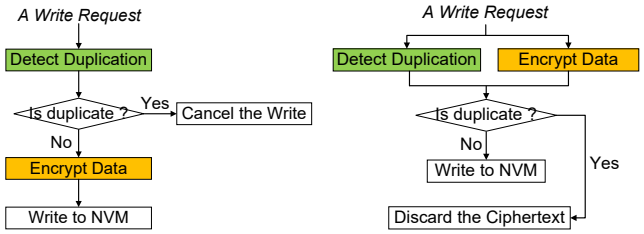
(e.g., DCW [9] and FNW [26]) ineffective [22], [23]. DeWrite instead exploits line-level data deduplication to improve the endurance and performance of secure NVMM.

A number of existing works [48]–[51] demonstrate that CPU caches contain abundant cache-line-level data duplications. These duplications are generally produced by the program behaviors such as copying and assignment [48], similar/duplicate data inputs [52], and memory initialization [21]. The fact that the data in CPU caches are read from main memory motivates us to consider whether abundant line-level duplications also exist in main memory. If true, we can directly cancel the writes of duplicate cache lines, which significantly improves the performance and endurance of NVMM. To explore how many cache lines written to main memory are duplicate, we have examined 20 applications from SPEC CPU2006 [27] and PARSEC 2.1 [28] benchmark suites. During executing these applications, we measure the number of duplicate lines written to NVMM, as shown in Figure 2.

Figure 2 shows that the percentages of duplicate lines vary from 18.6% to 98.4% across the 20 applications. We also observe that in only one application, i.e., sjeng, the duplicate lines are dominated by zero lines. The duplicate lines in other applications are mostly non-zero lines. Related work Silent Shredder [21] proposed to eliminate zero-content lines to reduce the writes to NVMM, which in our experiments reduces on average 16% of the writes. In comparison, eliminating all duplicate lines via line-level deduplication results in much higher write reductions for the applications, which is 58% on average. Hence, in DeWrite, we study the use of deduplication for enhancing the endurance and performance of secure NVMM, which is detailed in the rest of the paper.

III. The Design of DeWrite

Directly performing encryption and deduplication in the NVMM incurs two important issues. First, the latency of memory encryption and duplication detection are brought into the critical path of memory writes, which significantly decrease the system performance. This is because memory writes in the NVMM are on the critical path of application execution [15]–[18], which is different from traditional volatile memory. In traditional DRAM-based main memory, a processor can append a write in the write queue (WRQ) and further issue the next write without waiting for the completion of the previous one [53]. Thus writes are usually off the critical path



(a) The direct way (b) The parallel way

Fig. 3: Integrating deduplication and encryption.

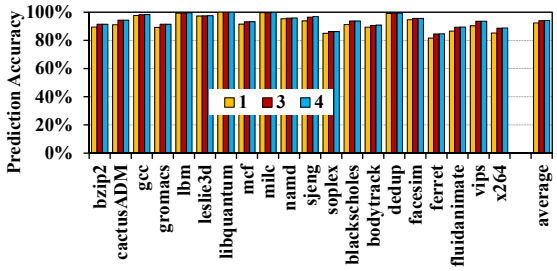


Fig. 4: The prediction accuracy. *(The '1', '3', and '4' indicate the use of 1, 3, and 4 most recent memory writes respectively.)*

of application execution due to the use of WRQ and do not incur processor stall cycles. In contrast, by using NVMs as persistent memory, the ordering of memory writes has to be ensured for consistency guarantee via cache line flushes and memory fences [12], [13], [54]–[56]. Thus the processor has to stall and wait for a memory write to be completed before issuing the next one. Thus memory encryption and duplication detection incur more processor stall cycles as they need to be performed before writing cache lines. Second, both NVM encryption and deduplication cause heavy metadata overheads. Counter mode encryption requires metadata such as the per-line counters. At the same time, to eliminate duplicate lines, deduplication also requires metadata including the per-line hashes and address mappings as shown in Section III-B. The storage and processing of these metadata incur substantial time and space overheads.

To address these issues, DeWrite judiciously integrates NVM encryption and deduplication by proposing a prediction-based parallel scheme (§ III-A) to improve the system performance, as well as a co-located metadata storage scheme (§ III-C) to reduce the metadata space overhead. Our proposed lightweight deduplication scheme is presented in § III-B.

A. Prediction-based Parallelism between Dedup & Encryption

In the traditional secure NVMM solutions [21]–[23], the cache lines are first encrypted and then written to NVMM. DeWrite leverages deduplication to reduce the number of writes to the secure NVMM by eliminating the writes of duplicate cache lines. The direct way to perform deduplication on the encrypted NVMM is shown in Figure 3a. For a cache line to be written to the NVMM, DeWrite first detects duplication. If the duplication exists in the NVMM, it cancels the write of the cache line and stores the address mapping relationship between the eliminated cache line and its duplicate in the NVMM into an address mapping table (presented in Section III-B). Otherwise, the cache line is encrypted and written to the NVMM. The former reduces the write latency by cancelling the write. The latter however increases write latency since the duplication detection and data encryption are executed serially in the critical path of the memory write. Thus the direct way is inefficient for applications where most lines are non-duplicate, such as `bzip2` and `vips`.

To avoid the serialization of detecting duplication and encrypting data, we consider a parallel way to perform deduplication on encrypted NVMM. As shown in Figure 3b, for a cache line to be written to the NVMM, cache line encryption and duplication detection are carried out in parallel. If no duplicates exist in the NVMM, the encrypted cache line is directly written to the NVMM. Otherwise, the encrypted cache line is discarded. However, for applications where most cache lines are duplicate, such as `cactusADM` and `lbm`, the parallel way becomes inefficient since the encryption is unnecessary and causes extra computation (and energy consumption) overhead from the AES circuit [57].

In summary, the direct way is efficient for the duplicate cache lines but inefficient for the non-duplicate cache lines. The parallel way has the opposite effects. Intuitively, the best solution is to use the direct way for duplicate cache lines and the parallel way for non-duplicate cache lines, respectively.

To address the problem of identifying whether a cache line is duplicate beforehand, we propose a simple yet effective prediction scheme by exploiting the duplication states of the most recent memory writes recorded using a history window. Specifically, DeWrite maintains an on-chip history window for the whole main memory. The history window records the duplication states of the most recent cache lines written into main memory. If the most recent memory writes are mostly duplicate (or non-duplicate), the next memory write is predicted to be duplicate (or non-duplicate). The rationale comes from our observation that the duplication states of cache lines written to main memory have temporal locality, i.e., duplicate (or non-duplicate) memory writes are usually consecutive. As shown in Figure 4, the duplication states of average 92% memory writes are the same as those of their previous ones. This observation can be interpreted that if a cache line written to main memory is duplicate (or non-duplicate), its next cache line written to main memory is also duplicate (or non-duplicate) with 92% probability.

The history window records the duplication state of only the previous one memory write, achieving about 92% prediction accuracy. In order to improve prediction accuracy, we further record the duplication states of the multiple most recent memory writes in the history window. We give some examples in the following. We first analyze the use of two most recent memory writes, which has four cases, i.e., '0 0', '0 1', '1 0', and '1 1'('1' and '0' respectively correspond to duplicate and non-duplicate writes). We observe that the prediction results of using the two most recent memory writes are the same as those of using the previous one memory write. Furthermore, the use of the three most recent memory writes produces two cases.

TABLE I: Traditional Deduplication v.s. DeWrite

(a) Comparisons of hash computation latency and sizes

Hash Functions	SHA-1	MD5	CRC-32
Latency	321 ns	312 ns	15 ns
Size	160 bits	128 bits	32 bits

(b) Comparisons of duplication detection latency

Methods	Traditional	DeWrite
A duplicate line	$\geq$ 312 ns + t_Q	91 ns + t'_Q
A non-duplicate line	$\geq$ 312 ns + t_Q	15 ns + t'_Q

If the number of '1' is larger than that of '0' in the three most recent memory writes, the prediction result is '1'. Otherwise, the result is '0'. The prediction accuracy is shown in Figure 4. The use of the three most recent memory writes improves on average 1.5% of accuracy (from 92.1% to 93.6%) compared with using one. We also evaluated the accuracy of using more than three ones, which leads to negligible increase in accuracy. DeWrite hence only needs to record the duplication states of three most recent memory writes to NVMM, and thus the storage overhead of the history window is 3 bits.

In summary, DeWrite maintains a 3-bit history window for the whole main memory to predict whether a cache line to be written into NVMM is duplicate. If a cache line is predicted to be non-duplicate, DeWrite encrypts data in parallel with duplication detection to reduce write latency. Otherwise, DeWrite detects duplication without encrypting data to reduce energy overhead from encryption.

B. Light-weight Deduplication for NVMM

We first present an overview of light-weight in-line deduplication for NVMM in DeWrite. We then present the data structures for supporting the in-line deduplication.

1) An Architecture Overview: To perform in-line deduplication, DeWrite leverages the asymmetric property of NVMs that write latency is much higher than read latency (i.e., $3 \sim 8\times$) [10], [21], [29]. To detect duplicate cache lines, DeWrite computes a light-weight hash to summarize the contents of cache lines, rather than the cryptographic hash with high computation latency. If the hash of the cache line matches that of an existing line in NVMM, DeWrite reads the line and compares the corresponding data byte by byte. Thus DeWrite eliminates a duplicate write at the cost of a read.

We compare the latency of duplication detection between traditional deduplication and DeWrite in Table I. Traditional deduplication includes existing main memory deduplication and external storage deduplication as discussed in Section V, which use a cryptographic hash function (e.g., SHA-1 [38] and MD5 [39]) to compute the fingerprints of data and assume no hash collisions. Thus the data are considered to be duplicate if their fingerprints are identical. The detection latency using the cryptographic hash function, regardless of the duplication of the cache line, is more than 312 ns + t_Q, where t_Q denotes the latency of querying the fingerprint store, as shown in Table Ib. The duplication detection latency is even higher than the NVM write latency (300 ns). Hence, traditional deduplication is not cost-effective to eliminate cache-line-level duplications.

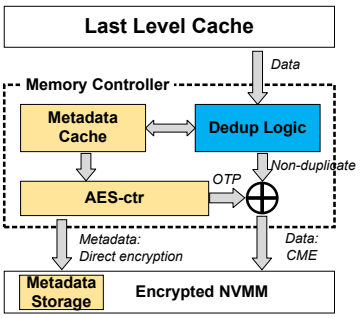


Fig. 5: The hardware architecture of DeWrite. *(We use the counter storage and counter cache in existing secure NVMMs [21]–[23] respectively as the metadata storage and metadata cache. The dedup logic is the new component.)*

Instead, DeWrite uses the light-weight hash, i.e., CRC-32, where hash collisions are practically unavoidable. Thus, if the hash of the cache line matches that of an existing line in NVMM, DeWrite reads the line and compares the corresponding data. Only when the data are identical, the cache line is considered to be duplicate. The latency of computing a CRC-32 hash is 15 ns and the latency of reading a line is 75 ns [21], [22], as the configurations shown in Section IV-A. The comparison of two lines can be implemented in hardware logic with low latency [9], i.e., 1 cycle. Hence, if a cache line is duplicate, the latency to detect the duplicate cache line in DeWrite is about 91 (=15+75+1) ns + t'_Q, where t'_Q denotes the latency of querying the hash store. If a cache line is non-duplicate, its hash value will not be found in the hash store and the read can be saved. Thus the latency to determine a non-duplicate cache line is 15 ns + t'_Q. Moreover, when a cache is employed to accelerate accesses to the hash store, the cache can store more CRC-32 hashes than SHA-1/MD5 hashes, because the former is much smaller. Hence, given the same size of cache, using CRC-32 hashes leads to a higher cache hit rate and correspondingly a lower access latency, i.e., $t'_Q < t_Q$. In summary, DeWrite significantly reduces the duplication detection latency by leveraging the read/write asymmetric property of NVMs and light-weight hashing.

The hardware architecture of DeWrite for supporting cache-line-level deduplication in the secure NVMM is shown in Figure 5. In order to reduce the metadata accesses to NVMM, existing work [21]–[23] on counter mode encryption extends the memory controller to include a write-back metadata cache used for buffering the counters. DeWrite uses this metadata cache to buffer the deduplication-related metadata and counters. The metadata persistency problem in the metadata cache is discussed in Section V. The required size of the metadata cache is evaluated in Section IV-E1. A memory region in the encrypted NVMM [21]–[23] is used to store the counters. We use this region to store the co-located metadata of deduplication and encryption, as presented in Section III-C. Hence, compared with existing work [21]–[23], the new component in DeWrite is the dedup logic that is used to determine whether a cache line is duplicate. The data are encrypted using counter mode encryption (CME). To avoid storing the

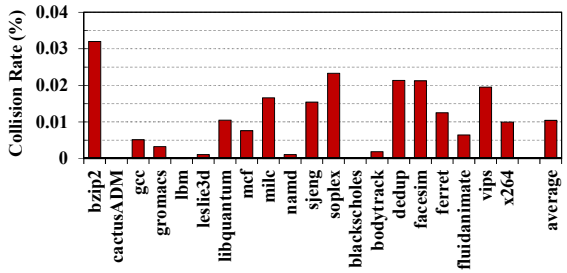


Fig. 6: The hash collision probability.

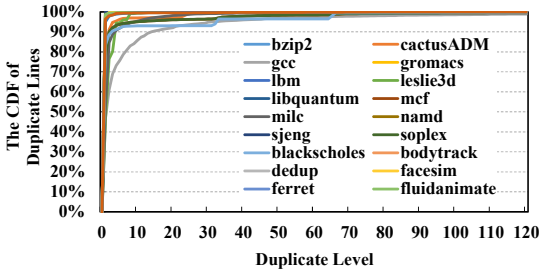


Fig. 7: The CDF of duplicate lines.

counters of the metadata, the metadata are encrypted using the direct encryption scheme presented in Section II-B. In this case, the latency of loading the metadata into the metadata cache increases since the decryption cannot be executed with memory access in parallel. However, it does not significantly affect the performance since the metadata cache miss ratio is very low as shown in Section IV-E2.

We consider the 256B of deduplication granularity to reduce the metadata overheads. Thus the sizes of a memory line in the NVMM and a cache line in the last level cache are 256B. The 256B cache line size is widely used in the existing work on NVMM [4], [8], [24], [29], [58], [59]. Moreover, the commercial processors, e.g., IBM z systems processors [60], [61], also use the 256B cache line size for all CPU caches.

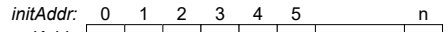
2) Data Structures for Supporting Deduplication: To support in-line deduplication, DeWrite uses four data structures for duplication detection and data management, including address mapping table, hash table, inverted hash table, and free space management table. These data structures are stored in the encrypted NVMM and the hot entries are maintained in the metadata cache. In the hash table, we reuse the prediction scheme presented in Section III-A to reduce NVM accesses. In the remaining three tables, we leverage the prefetching technique to improve the cache hit rates. The store space of these data structures is evaluated in Section IV-E1. In the following, we present the design of the four data structures.

• ***Address Mapping Management.*** For regular NVMM, the mapping between a cache line's address number and its storage location (i.e., rank/bank/row/column) is one-to-one. The storage location is directly computed by the address number. However, when using deduplication, the relationship between the address number and the storage location becomes many-to-one since duplicate lines are removed from the initial locations. Hence, DeWrite uses an address mapping table to maintain the many-to-one mapping relationships. Each entry in the address mapping table denotes a `<initAddr,realAddr>` pair. The `realAddr` denotes the address number of the line storing the real data. The `initAddr` denotes the initial address number of the referenced line that contains the same data as the line stored at `realAddr`. The address mapping table is a data structure of sequential storage, e.g., a one-dimensional array. The address mapping table stores only the real addresses, since the corresponding initial addresses can be denoted by their locations in the table.

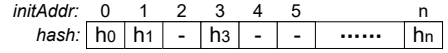
DeWrite stores the address mapping table in the encrypted NVMM and maintains the recently-accessed address mappings in the metadata cache. The sequential storage in the address mapping table retains the space locality of initial addresses. When an address mapping is read, the following address mappings are also prefetched to the metadata cache. For example, if the data size in each read is 256B and the size of the real address is 4B, an NVM access prefetches 64 sequential `<initAddr,realAddr>` pairs. The storage structure reduces the NVM accesses when querying the address mappings due to the access locality.

• ***Hash Table for Duplication Detection.*** In order to quickly locate the duplicate lines in NVMM, DeWrite maintains a hash table, in which each entry contains three items, i.e., `<hash,realAddr,reference>`. We use the 32-bit hash values to index the entries, where each entry stores the address of the corresponding line and its reference count. The `reference` denotes the number of initial addresses that are mapped to the `realAddr`. Each hash entry possibly contains multiple values due to hash collisions. However, the collision probability is extremely low (less than 0.01% on average), as shown in Figure 6. The reference is 8 bits, which is sufficiently large for counting the references of lines, since more than 99.999% lines have a reference that is smaller than 255, based on our experimental results in Figure 7. In the rare cases where a reference reaches 255, we consider the corresponding line as a highly referenced line and no longer increase its value. When a new cache line is duplicate with a highly referenced line, DeWrite does not process it as a duplicate and avoids overflowing the reference.

DeWrite stores the hash table in the encrypted NVMM and maintains recently-accessed hash entries in the metadata cache. When there is no matching entry in the cache for a hash query, a memory access is required to query the hash table in the NVMM. However, if still no match in the hash table, the latency becomes high due to the extra latency of memory access. In order to reduce the NVM accesses during the hash query, we leverage a prediction-based NVM access (PNA) scheme which reuses the prediction scheme described in Section III-A. If the hash query for a cache line finds no match in the cache, only when the prediction result is duplicate, DeWrite further queries the in-NVM hash table. Otherwise, DeWrite directly treats the cache line as non-duplicate and carries out writes to NVM without further querying the in-NVM hash table. Due to the high prediction accuracy as shown in Figure 4, the PNA scheme eliminates most unnecessary NVM accesses from the duplication detection for non-duplicate cache lines. Our evaluation in Section IV-B also confirms this performance

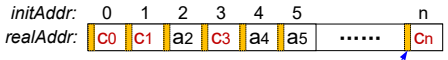


(a) The address mapping table

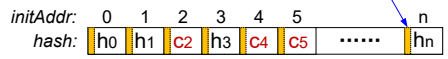


(b) The inverted hash table

Fig. 8: The metadata storage of deduplication. *(The memory lines with the initAddr 2, 4, and 5, are deduplicated. '$a_\sharp$': the real address; '$h_\sharp$': the hash value; '−': null.)*



(a) The address mapping table



(b) The inverted hash table

Fig. 9: The co-located metadata storage between deduplication and encryption. *('$c_\sharp$': the counter.)*

improvement scheme does not comprise DeWrite's enhancement on endurance.

• ***Inverted Hash Table for Stale Hash Cleaning.*** When a memory line is rewritten, the hash value of the old data in the line should be removed from hash table. An inverted hash table is required to query the stale hash entry in hash table based on the line address. The inverted hash table maps the addresses of memory lines to their hash values, in which each entry denotes a `<initAddr,hash>` pair. DeWrite stores the inverted hash table in the encrypted NVMM and maintains the recently-accessed entries in the metadata cache. The LRU replacement policy is used for the data eviction in the metadata cache. The inverted hash table is stored sequentially by the real addresses and prefetching is used to reduce NVM accesses like the address mapping table.

• ***Free Space Management.*** Due to deduplication, when a new write arrives, the old data in the corresponding line stored in NVMM is possibly referenced by one or more other initial addresses. In this case, the old data in the line cannot be overwritten and the system needs to find a free location to write the new data. A free space management (FSM) table is used to manage the free lines in the NVMM. A one-bit flag is used to label whether a line is free. The FSM table is also a data structure of sequential storage maintained in the NVMM and the recently-accessed entries are buffered in the metadata cache. Since the flag of each initial address is only one bit, one memory access can prefetch the flags of a large number of sequential initial addresses.

C. Metadata Colocation between Dedup & Encryption

Existing counter mode encryption needs to store per-line counters to encrypt/decrypt data. The per-line counter is 28 bits for each line [22]. Deduplication also produces some metadata. To reduce the space overhead of metadata, we propose a co-located metadata storage scheme between deduplication and encryption via embedding the per-line counters into the data structures for deduplication.

Figure 8 shows the initial address mapping table and inverted hash table for deduplication. There are two kinds

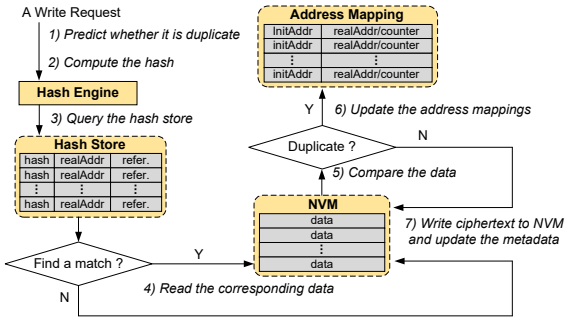


Fig. 10: The workflow of a write operation.

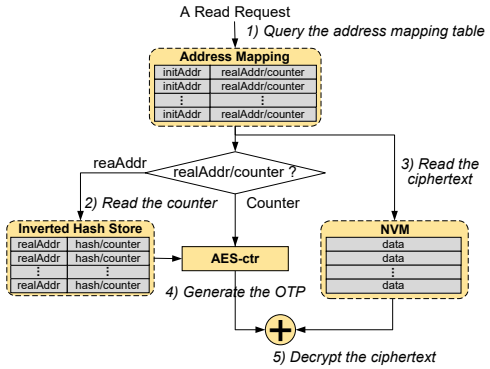


Fig. 11: The workflow of a read operation. *(Step 2 and Step 4 can be executed in parallel with Step 3.)*

of memory lines in the NVMM, i.e., deduplicated lines and non-deduplicated lines. For the deduplicated lines, e.g., the memory lines with the `initAddr` 2, 4, and 5 in Figure 8a, the real addresses containing their data contents are stored in the address mapping table. But their corresponding locations in the inverted hash table are null as shown in Figure 8b, since the deduplicated lines does not contain valid data. For the non-deduplicated lines, e.g., the memory lines with the `initAddr` 0, 1, and 3 in Figure 8a, their storage locations in the address mapping table are null due to no deduplication, and their hash values are stored in the inverted hash table. Therefore, we observe that for each memory lines in the NVMM, either its `realAddr` location or its hash location is null.

Based on the above observation, we embed the counter of each memory line into the null location either in the address mapping table or in the inverted hash table, as shown in Figure 9. Moreover, we use 1-bit flag in each location in the two tables. In the address mapping table, the flag bit is used to distinguish whether the location stores a real address or a counter, and in the inverted hash table, the flag bit is used to distinguish whether the location stores a hash or a counter. As a result, the traditional counter table is removed, reducing the metadata storage overhead.

Finally, we present the main workflows of write and read operations in DeWrite, as shown in Figures 10 and 11.

IV. Evaluation

A. Methodology

As real hardware is not available for implementing NVMM and the proposed deduplication and encryption techniques,

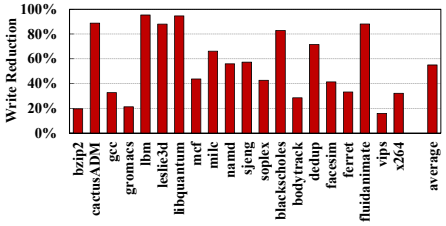


Fig. 12: The savings on writes to NVMM.

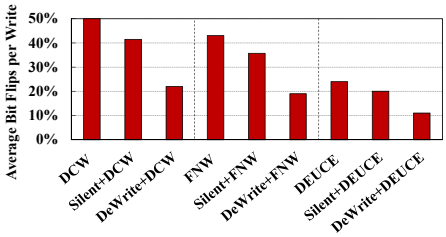


Fig. 13: The average bit flips per write using different techniques.

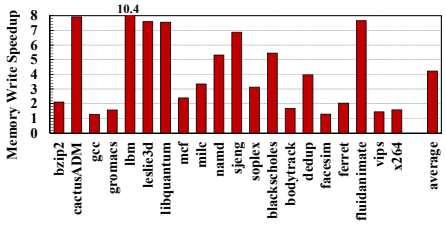


Fig. 14: The write speedup compared with the traditional secure NVMM.

we use gem5 [43] with NVMain [44] to evaluate DeWrite. NVMain is a main memory simulator for emerging NVM technologies, which can accurately simulate the timing and energy information of NVM systems. We have implemented a DeWrite prototype including deduplication and encryption modules in NVMain. We compare DeWrite with the traditional secure NVM system that uses the counter mode encryption without deduplication. The configuration parameters are shown in Table II. This system consists of a four-level cache hierarchy, following the expected trend of modern architecture [21], [62]. The size of all CPU cache lines is 256B, which is widely used in existing work [4], [8], [24], [29], [58], [59] and real systems [60], [61]. The metadata cache's size is 2MB, to maintain the recently-accessed counters in the traditional secure NVM system, and the recently-accessed metadata in DeWrite. Moreover, without loss of generality, we model PCM technologies [63] to evaluate DeWrite that in fact can be also used in other NVMs. The PCM is modeled like Xu et al.'s work [64]. Since NVMain does not include encryption-related configurations, we set the latency of AES encryption to 96ns per line and the energy overhead of AES encryption to 5.9nJ per 128-bit block based on the specifications [65]–[67].

We evaluate DeWrite on both single-threaded and multiple-threaded applications from SPEC CPU2006 [27] and PARSEC 2.1 [28] benchmark suites. The two benchmark suites contain rich and diverse real-world applications chosen from many different areas, e.g., computer vision, enterprise servers, physics computing, artificial intelligence, enterprise storage, etc, which are widely used to evaluate the performance of memory systems [4], [9], [21]–[23], [58]. The SPEC CPU2006 benchmarks are single-threaded and run with the *ref* input set. To evaluate the performance on multiple-threaded benchmarks, we run 8 applications from PARSEC 2.1 with the *simlarge* input set, where the number of threads is set to 4. We configure all the applications with the default settings. We first warm up the system caches for 10 million instructions, and then run each application for 4 billion instructions.

TABLE II: The configurations of the NVM system.

Processor	
CPU	4 cores, X86-64 processor, 2GHz
Private L1 cache	32KB, 8-way, LRU, 2-CPU-cycle latency
Private L2 cache	128KB, 8-way, LRU, 8-CPU-cycle latency
Shared L3 cache	2MB, 8-way, LRU, 25-CPU-cycle latency
Shared L4 cache	32MB, 8-way, LRU, 50-CPU-cycle latency
Main Memory Using PCM	
Capacity	16GB, (16 banks, distributed in 2 ranks)
Read/write latency	75ns/300ns
Metadata cache	2MB, LRU, 25-CPU-cycle latency

In the rest of this section, we show the improvements made by DeWrite in endurance, performance, and energy consumption, as well as its overhead.

B. NVM Endurance

One of main objectives of DeWrite is to reduce the number of writes to NVMM and enhance its endurance by identifying and eliminating the writes of duplicate cache lines. As shown in Figure 12, we observe that DeWrite reduces on average 54% of whole-line memory writes across all applications. For some applications which contain a large number of duplicate writes, e.g, `cactusADM`, `libquantum`, `lbm`, and `blackscholes`, DeWrite even reduces more than 80% of whole-line memory writes. Nevertheless, the memory writes reduced by DeWrite (on average 54%) is less than the total number of duplication lines existing in these applications (on average 58%), as shown in Figure 2. About 4% of write reduction is missed due to two reasons. First, due to the prediction-based NVM access scheme and the limited range of the hash table references (Section III-B2), DeWrite fails to detect a small number of duplicate lines, i.e., on average 1.5%. Moreover, the dirty data evicted from the metadata cache incur on average 2.6% extra writes. The amount of metadata writes to the NVMM is small due to the high hit rate (over 98%) achieved by the metadata cache, which is discussed in Section IV-E.

As DeWrite is a line-level write reduction technique for eliminating the whole-line writes of duplicate lines, it can be combined with the bit-level write reduction techniques, such as DCW [9], FNW [26] and DEUCE [22], for reducing the bit writes of non-duplicate lines. We compare DeWrte with these state-of-the-art write reduction techniques in terms of the average bit flips per write, as shown in Figure 13. We observe that DCW and FNW cause high bit-flip ratios, i.e., 50% and 43% due to the diffusion property of data encryption. DEUCE reduces the bit-flip ratio to 24% by using partial-line re-encryption. Silent Shredder [21] is a line-level write reduction technique which aims to eliminate the writes of full-zero cache lines from data shredding. However, since the full-zero lines are not common, i.e., about 16% on average, as shown in Figure 2, Silent Shredder reduces only a small number of bit flips when combined with the bit-level techniques. In contrast, DeWrite eliminates the writes of duplicate lines that are abundant besides full-zero lines. When combined with DCW, FNW, and DEUCE, DeWrite reduces the average bit

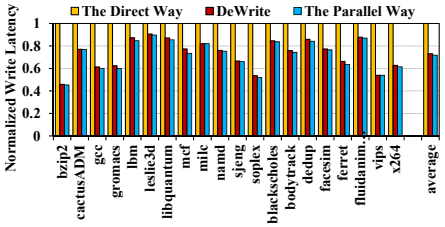


Fig. 15: The write latency normalized to that of the direct way.

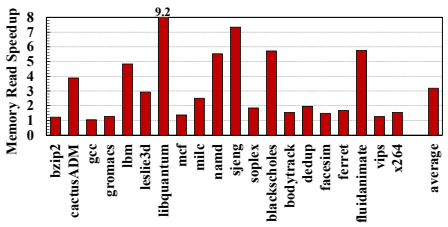


Fig. 16: The read speedup compared with the traditional secure NVMM.

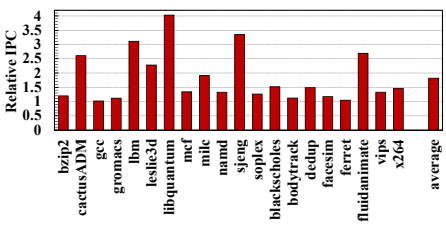


Fig. 17: The relative IPC compared with the traditional secure NVMM.

flips from 50% to 22%, from 43% to 19%, and from 24% to 11%, respectively. Therefore, based on these bit-level write reduction techniques, DeWrite further reduces over half of bit flips, and thus increases the NVM lifetime substantially.

C. System Performance

1) Write Speedup: DeWrite reduces the write latency from two aspects. First, DeWrite removes the write latency of duplicate writes. Second, DeWrite eliminates duplicate writes, which also reduces the write latency of other non-duplicate writes to the same bank via decreasing their wait time. The speedup ratio of memory write is interpreted as the write latency of the traditional secure NVM system without deduplication divided by that of DeWrite, as shown in Figure 14. We observe that DeWrite achieves an on average 4.2× speedup ratio in memory writes across 20 applications. The speedup ratio of memory writes depends on the fraction of the number of eliminated memory writes in the applications. In some applications with high write reduction ratios (e.g., `cactusADM` and `lbm`), the write speedup ratios are up to 8×.

In order to show the benefits from the prediction-based parallel scheme in DeWrite, we compare the write latencies of the direct way, the parallel way and DeWrite. The write latencies of the three schemes are normalized to that of the direct way, as shown in Figure 15. We observe that the parallel way has the lowest write latency due to executing duplication detection and encryption in parallel for all writes. DeWrite executes duplication detection and encryption in parallel for the writes which are predicted to be non-duplicate. DeWrite obtains nearly the same write latency as the parallel way due to the high prediction accuracy. Nevertheless, DeWrite significantly reduces the energy overhead compared with the parallel way as presented in Section IV-D. The direct way has the highest write latency. Compared with the direct way, DeWrite reduces 27% write latency on average.

2) Read Speedup: DeWrite reduces the wait time of read requests to improve read performance by eliminating duplicate write requests. But to handle a single read request, DeWrite needs to first query the address mapping table and then read the actual data, which slightly increases the read latency. Nevertheless, the increased latency of accessing address mapping table appears negligible, compared with the reduced read latency from eliminating duplicate writes. The speedup ratio of memory read is interpreted as the average read latency of the traditional secure NVM system divided by that of DeWrite, as shown in Figure 16. We observe that DeWrite achieves an average 3.1× read speedup ratio across 20 applications.

3) Instructions per Cycle: Due to the elimination of duplicate writes, the write and read latencies are reduced, which improves the overall performance of the system, i.e., instructions per cycle (IPC). Figure 17 shows the relative IPC of DeWrite normalized to the traditional secure NVM system. We observe that DeWrite achieves on average 82% IPC improvement across all applications.

4) The Worst-case Performance: DeWrite exploits the deduplication to improve the performance. When there are no duplicate cache lines written to NVMM which in fact rarely occurs, DeWrite incurs slight acceptable decrease of the system performance. To investigate the performance of DeWrite in the worst case that no duplicate writes exist, we generate a benchmark by inserting the randomized values into a two-dimensional array and then traversing the array. Thus there are no duplicate cache lines written to NVMM in this benchmark. Figure 18 shows the write latency, read latency and IPC normalized to the traditional secure NVMM when running this benchmark.

We observe that DeWrite incurs negligible performance degradation in the worst case, compared with the traditional secure NVMM. The reason is that: 1) For the write latency, since there are no duplicate writes, the prediction-based parallel scheme in DeWrite always executes the encryption and duplication detection in parallel, and thus removes the duplication detection latency off from critical path of memory writes. Moreover, the writes of deduplication-related metadata are mostly buffered in the metadata cache. Hence, DeWrite only slightly increases the write latency. 2) For the read latency, since there are no duplicate writes, all counters are stored in the address mapping table in DeWrite as presented in Section III-C. DeWrite slightly increases the read latency due to first querying the counter in the metadata cache and then reading the data. 3) As a result, DeWrite incurs negligible decrease on the IPC, i.e., less than 3% decrease.

D. Energy Consumption

As writes in NVMs consume significant energy, DeWrite eliminates a large number of duplicate writes and has a great impact on the energy consumption of the system. We measure energy consumption of the secure NVM system including NVM, AES circuit and dedup logic. We compare the energy overhead of DeWrite and the traditional secure NVM system

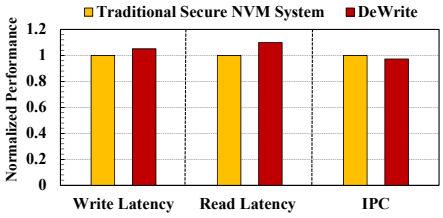


Fig. 18: The normalized performance in the worst case.

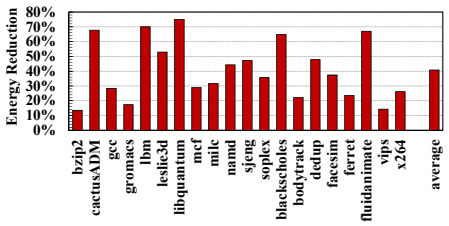


Fig. 19: The energy savings of DeWrite.

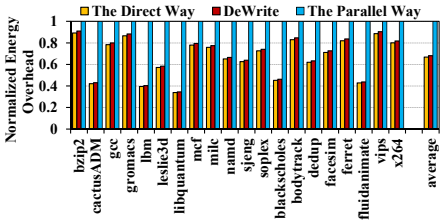


Fig. 20: The energy overheads normalized to that of the parallel way.

without deduplication, as shown in Figure 19. We observe that DeWrite reduces 40% of energy overhead on average across all applications. The reason is that DeWrite reduces the number of writes and data encryption. The energy overhead of the dedup logic including CRC-32 hash computation and cache line comparison is negligible, compared with that of AES [67].

To show the benefits from the prediction-based parallel scheme in DeWrite, we also compare the total energy consumptions of the direct way, DeWrite, and the parallel way. The energy overheads of the three schemes are normalized to that of the parallel way, as shown in Figure 20. The direct way has the lowest energy overhead due to only encrypting non-duplicate writes. DeWrite encrypts the writes that are predicted to be non-duplicate during duplication detection. DeWrite obtains nearly the same energy overhead as the direct way due to the high prediction accuracy. The parallel way encrypts all writes and hence has the highest energy overhead. Compared with the parallel way, DeWrite reduces 32% energy overhead on average.

According to the results shown in Figures 15 and 20, we observe that the direct way has the highest write latency and the lowest energy overhead. The parallel way achieves the opposite results. DeWrite has nearly the same energy overhead as the direct way while reduces the 27% write latency on average; it has nearly the same write latency as the parallel way while reduces 32% energy overhead on average. Therefore, DeWrite is much better in terms of both the write speedup and energy reduction by leveraging the prediction-based parallel scheme with high prediction accuracy.

E. Space Overheads of the Metadata Storage and Cache

1) The Metadata Storage: The metadata storage includes four tables, i.e., the address mapping table, inverted hash table, hash table, and FSM table. In the inverted hash and address mapping tables, the storage overhead is 4B+1bit/line, due to the 4B real address/hash and 1bit flag in each entry. The 4B real address can address up to 1TB of NVM with 256B line size, which is sufficiently large for the 16GB NVM in our configurations. In the hash table, each entry is 9B and the number of entries is related with the deduplication ratio. Since the deduplication ratios range from 18.6% to 98.4% across the 20 applications as shown in Section II-C, the storage overhead in the hash table is less than $9 * (1 - 18.6\%) < 8$B/line. The storage overhead of FSM table is 1 bit/line. Hence, the total storage overhead of metadata is (4B+4B+8B+3bit)/256B $\approx 6.25\%$ of the NVM capacity.

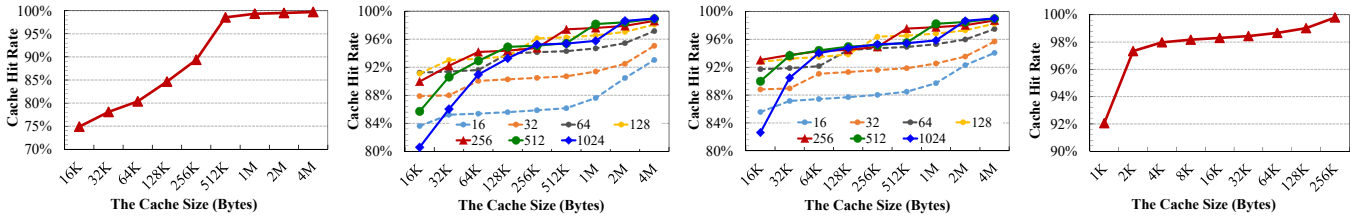
The approximate 6.25% metadata storage overhead of DeWrite is lower than that of DEUCE [22]. DEUCE has two kinds of metadata. First, DEUCE needs a 1-bit flag per word in each line to indicate whether the word is modified. Each word is 16 bits in DEUCE and hence the storage overhead of the flags is 1 bit/ 16 bits = 6.25%. Second, DEUCE uses the counter mode encryption in which the storage overhead of the per-line counter is 28 bits/line. DeWrite only has the 6.25% metadata storage overhead from deduplication while does not have the metadata storage overhead from memory encryption since our proposed co-located metadata storage scheme reduces the storage overhead from counters.

2) The Metadata Cache: DeWrite maintains a metadata cache to reduce metadata accesses to NVMM. The metadata cache should be as small as possible while maintaining a high hit rate. We investigate the influence of the cache size on the average cache hit rate across the 20 applications.

For the hash table cache, as shown in Figure 21a, we observe that increasing the cache size beyond 512KB has little impact on the cache hit rate. For the address mapping and inverted hash tables, the prefetching granularity also impacts on the hit rate, and we investigate the cache hit rates with different prefetching granularities, from 16 to 1024 entries on each NVM access. Figures 21b and 21c show that the size of 512KB with the prefetching granularity of 256 achieves high cache hit rates for both address mapping cache and inverted hash cache. Larger cache sizes result in only small increase of hit rate. Considering the tradeoff between the cache size and hit rate, we configure both caches with a capacity of 512KB. For the FSM table, as shown in Figure 21d, its cache hit rate achieves 98% for only 4KB cache size and reaches over 99% for 128KB cache size. This is because the flag of each initial address is only one bit and thus the size of the FSM table is very small. We configure the cache of the FSM table with a capacity of 128KB. Therefore, the required total capacity of the metadata cache is 512KB*3+128KB = 1664KB $<$ 2MB.

V. Related Work

Secure Non-volatile Main Memory. As both write endurance and security are important problems for NVMM, many schemes have been proposed to reduce writes in the encrypted NVMM. i-NVMM [68] proposed that the hot data are kept in the unencrypted form in the memory for improving the system performance and encrypted only when the system is powered down. However, i-NVMM fails to protect the memory against the bus snooping attack, since the hot data



(a) Hash table cache (b) Address mapping cache (c) Inverted hash cache (d) Free space management cache

Fig. 21: The cache hit rates of different caches with different sizes.

are unencrypted through the bus. Unlike i-NVMM, DeWrite encrypts both hot and cold data written to NVMM and defends against both the stolen DIMM and bus snooping attacks. DEUCE [22] reduces the bits written to secure NVMM by only re-encrypting modified words (i.e., 2 bytes) in a cache line and keeping unmodified words in the last encrypted state. Based on DEUCE, SECRET [23] focuses on MLC NVMs and further reduces the re-encryption of full-zero words in a cache line. DeWrite is a line-level write reduction scheme, which is orthogonal with the subline-level (word-level) ones, like DEUCE and SECRET. Based on these subline-level write reduction techniques, DeWrite can further reduce over half of bit flips, as evaluated in Section IV-B. Awad et al. [21] proposed Silent Shredder, which eliminates the writes of full-zero cache lines from data shredding. Unlike Silent Shredder, DeWrite eliminates all duplicate lines besides full-zero lines.

For ensuring the metadata consistency and persistence in the metadata cache on a system failure (e.g., power failure and system crash), Silent Shredder [21] uses a battery-backed write-back metadata cache. Liu et al. [69] introduce a new programming primitive `counter_cache_writeback()` to enable programmers to actively write back the metadata in the metadata cache into the write queue when necessary, and leverages the asynchronous DRAM refresh (ADR) [70]–[72] to persist the data in the write queue on a system failure. SecPM [73] proposes a counter cache write-through scheme to guarantee crash consistency of counters. Moreover, NVMs, e.g., STT-RAM, can be used to achieve a non-volatile metadata cache [74] to ensure the metadata persistence. These metadata persistence schemes can be also used in DeWrite.

Main Memory Deduplication. Memory deduplication [30]–[33], [75] has been studied to save DRAM space by identifying and merging identical memory pages. DeWrite is different from memory deduplication in the following aspects. First, in memory deduplication, the duplicate pages are first written into the memory and then identified and merged in the background, which hence fails to reduce the writes to memory. DeWrite reduces the writes by identifying and eliminating duplicate cache lines before writing them to NVMs. Second, memory deduplication eliminates duplicate pages, which is effective only for special applications, e.g., virtual machines, which have rich page-level redundancies. DeWrite aims to eliminate line-level redundancies which generally exist in real-world applications as shown in Section II-C. Moreover, traditional memory deduplication suffers from the side channel attacks when the attacker and victim run their applications in the same computer [76]–[78]. The side channel attacks are beyond the scope of this paper that aims to protect NVMM from physical access based attacks in which the attacker is not the user of the computer as presented in Section II-A.

Storage Deduplication. Deduplication has been widely used in storage systems [34]–[37], [79] to reduce the storage overhead of external memories, e.g., disks and SSDs. For example, CAFTL [80] and CA-SSD [81] deploy deduplication in the flash translation layer of SSDs for enhancing their lifespan. Moreover, NV-Dedup [82] employs deduplication in NVM-oriented file systems to eliminate data redundancy at the chunk level (4KB size) for saving NVM space and improving performance. However, these works are fundamentally different from DeWrite. Because they use cryptographic hash functions, e.g., SHA-1 [38] and MD5 [39], to compute the fingerprints of data. Thus duplication can be detected by comparing fingerprints. In fact, the latency of cryptographic hashing, typically more than 300 ns [40], [41], makes it unsuitable for highly latency-sensitive main memory to eliminate cache-line-level duplications, as analyzed in Section III-B. Unlike them, DeWrite proposes a cost-efficient cache-line-level deduplication technique by leveraging the intrinsic read/write asymmetry of NVMs and light-weight hashing, significantly reducing the latency of detecting duplication.

VI. Conclusion

In this paper, we propose DeWrite to enhance lifetime and performance of secure NVMM through deduplicating writes. DeWrite addresses the challenges of performing in-line deduplication on secure NVMM and judiciously integrating deduplication and NVM encryption to deliver high performance. DeWrite is implemented via revising the metadata store and metadata cache that already exist in secure NVMM and only adding a dedup logic into the memory controller, thus achieving the low design complexity. Our experimental results demonstrate that DeWrite eliminates 54% of writes to secure NVMM, and speeds up memory writes and reads by $4.2\times$ and $3.1\times$ on average. Meanwhile, DeWrite improves the IPC by 82% and reduces 40% of energy consumption on average.

Acknowledgements

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202, National Natural Science Foundation of China (NSFC) under Grant 61772212, and U.S. National Science Foundation awards CNS-1619653, CNS-1562837, and CNS-1629888.

REFERENCES

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, vol. 30, no. 1, pp. 131–141, 2010.

[2] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 4, pp. 1–134, 2011.

[3] Y. Xie, "Modeling, architecture, and applications for emerging memory technologies," *IEEE Design & Test of Computers*, vol. 28, no. 1, pp. 44–51, 2011.

[4] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *in Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[5] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," in *Proceedings of the 2013 IEEE 19th International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

[6] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.

[7] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[8] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "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.

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

[10] J. Yue and Y. Zhu, "Accelerating write by exploiting pcm asymmetries," in *Proceedings of the 2013 IEEE 19th International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

[11] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ecp, not ecc, for hard failures in resistive memories," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2010.

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

[13] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory." in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

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

[15] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: exploiting nvram in write-ahead logging," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[16] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DUDETM: Building durable transactions with decoupling for persistent memory," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

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

[18] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2011.

[19] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and high-performance memory control for persistent memory systems," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[20] K. David, P. Jeremy, and W. Tom, "AMD memory encryption," *AMD White Paper*, 2016.

[21] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "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.

[22] V. Young, P. J. Nair, and M. K. Qureshi, "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.

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

[24] J. Kong and H. Zhou, "Improving privacy and lifetime of pcm-based main memory," in *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010.

[25] W. Stallings, "Cryptography and Network Security (6th ed.)," pp. 67–68 (2014), 2014.

[26] S. Cho and H. Lee, "Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[27] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[28] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

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

[30] C. A. Waldspurger, "Memory resource management in vmware esx server," *in Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[31] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proceedings of the linux symposium (OLS)*, 2009.

[32] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao, "CMD: Classification-based memory deduplication through page access characteristics," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2014.

[33] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "XLH: more effective memory deduplication scanners through cross-layer hints," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013.

[34] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proceedings of the ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2001.

[35] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *Proceedings the USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[36] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Proceedings the USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[37] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "Cachededup: in-line deduplication for flash caching," in *Proceedings the USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[38] P. FIPS, "180-1. secure hash standard," *National Institute of Standards and Technology*, vol. 17, p. 45, 1995.

[39] R. Rivest, "The md5 message-digest algorithm," p. 1992, 1992.

[40] T. Kgil, L. Falk, and T. Mudge, "Chiplock: support for secure microarchitectures," *in Workshop on Architectural Support for Security and Anti-Virus*, 2004.

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

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

[43] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator,"

ACM SIGARCH Computer Architecture News, vol. 39, no. 2, pp. 1–7, 2011.

[44] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, 2015.

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

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

[47] M. Henson and S. Taylor, "Memory encryption: a survey of existing techniques," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 53, 2014.

[48] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level cache deduplication," in *Proceedings of the 28th ACM international conference on Supercomputing (ICS)*, 2014.

[49] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong, "Multi-execution: multicore caching for data-similar executions," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[50] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelgänger: a cache for approximate computing," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.

[51] M. Kleanthous and Y. Sazeides, "Catch: A mechanism for dynamically detecting cache-content-duplication and its application to instruction caches," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, 2008.

[52] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi, "Hicamp: architectural support for efficient concurrency-safe shared structured data access," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[53] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, "The virtual write queue: Coordinating dram and last-level cache policies," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010.

[54] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[55] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *Proceedings the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[56] P. Zuo, Y. Hua, and J. Wu, "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.

[57] S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni, "Midori: a block cipher for low energy," in *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2014.

[58] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "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.

[59] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers, "Improving write operations in mlc phase change memory," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.

[60] "IBM z systems microprocessor optimization primer," https://ibm.co/1qeGrpc, 2018.

[61] B. White, E. Bakker, P. Hamid, O. Lascu, F. Nogal, F. Packeiser, V. Ranieri, K. Stenfors, E. Ufacik, and C. Zhu, *IBM zEnterprise 196 Technical Guide*, ser. IBM redbooks. IBM Redbooks, 2012. [Online]. Available: https://books.google.com.hk/books?id=7RrCAgAAQBAJ

[62] J. Stuecheli, "Power8," in *Hot Chips*, 2013.

[63] Y. Choi, I. Song, M. H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y. J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y. T. Lee, J. Yoo, and G. Jeong, "A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth," in *Proceedings of the International Solid-State Circuits Conference (ISSCC)*, 2012.

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

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

[66] F. Huang, D. Feng, Y. Hua, and W. Zhou, "A wear-leveling-aware counter mode for data encryption in non-volatile memories," in *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[67] P. Hamalainen, T. Alho, M. Hannikainen, and T. D. Hamalainen, "Design and implementation of low-area and low-power aes encryption hardware core," in *9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, 2006.

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

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

[70] "Deprecating the PCOMMIT Instruction," https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, 2016.

[71] "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, 2017.

[72] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: a flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[73] P. Zuo and Y. Hua, "SecPM: a secure and persistent memory system for non-volatile memory," in *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[74] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[75] D. Skarlatos, N. S. Kim, and J. Torrellas, "Pageforge: a near-memory content-aware page-merging architecture," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[76] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Software side channel attack on memory deduplication," in *Poster session in ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[77] J. Xiao, Z. Xu, H. Huang, and H. Wang, "Security implications of memory deduplication in a virtualized environment," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[78] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security (EuroSec)*, 2011.

[79] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[80] F. Chen, T. Luo, and X. Zhang, "Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives." in *Proceedings the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[81] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing nand flash-based ssds." in *Proceedings the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[82] C. Wang, Q. Wei, J. Yang, C. Chen, Y. Yang, and M. Xue, "Nv-dedup: High-performance inline deduplication for non-volatile memory," *IEEE Transactions on Computers*, vol. 67, no. 5, pp. 658–671, 2018.