Scalable Crash Consistency for Secure Persistent Memory

Ming Zhang, Yu Hua[✉], Xuan Li, Hao Xu
WNLO, School of Computer, Huazhong University of Science and Technology

Abstract

Persistent memory (PM) suffers from data security and crash consistency issues due to non-volatility. Counter-mode encryption (CME) and bonsai merkle tree (BMT) have been adopted to ensure data security by using security metadata. The data and its security metadata need to be atomically persisted for correct recovery. To ensure crash consistency, durable transactions have been widely employed. However, the long-time BMT update increases the transaction latency, and the security metadata incur heavy write traffic. This paper presents Secon to ensure SEcurity and crash CONsistency for PM with high performance. Secon leverages a scalable write-through metadata cache to ensure the atomicity of data and its security metadata. To reduce the transaction latency, Secon proposes a transaction-specific epoch persistency model to minimize the ordering constraints. To reduce the amount of PM writes, Secon co-locates counters with log entries and coalesces BMT blocks. Experimental results demonstrate that Secon significantly improves the transaction performance and decreases the write traffic.

1 Introduction

Persistent memory (PM) offers salient features including non-volatility, high density and DRAM-like performance. The data in PM survive crashes, which however causes data remanence vulnerabilities, e.g., an attacker can obtain the data via stealing the PM DIMM or snooping the memory bus [2]. Counter-mode encryption (CME) [22] has been proposed to encrypt data on chip to protect the confidentiality. As counters can be replayed or tampered by attackers, the integrity trees, e.g., bonsai merkle tree (BMT) [19], further verifies data integrity to build secure PM systems. For each counter write, all layers of blocks in a BMT are updated. Any counter tampering can be detected by recomputing BMT blocks for value comparisons.

Moreover, since the data in PM are not lost after a crash, the data are required to be recovered to a consistent state (i.e., crash consistency). Unfortunately, the atomic write is only 8B in 64-bit CPUs [11]. To ensure that the data larger than 8B are atomically modified in PM, durable transactions with write-ahead-logging [12] are widely leveraged to guarantee that either "none" or "all" of a group of data are updated in PM. Furthermore, to correctly decrypt and verify data after a crash, it is critical to ensure that the data and its security metadata are atomically updated [13]. Otherwise, the data will be mistakenly decrypted by a mismatched counter.

Ensuring security and crash consistency becomes important for PM, which is however difficult to guarantee. An intuitive solution is to use transactions on secure PMs, but this incurs high overheads due to two reasons: 1) The long latency of BMT update exists on the critical path. Each write in a transaction is stalled until the BMT root of the log entry is updated, which increases the transaction execution latency. 2) The security metadata incur substantial memory writes. The requirement of atomically updating *data* and its security metadata, i.e., *counter* and *BMT blocks*, causes 3× writes, which shortens the PM lifetime and decreases system performance.

State-of-the-art designs fail to effectively guarantee data security and crash consistency for PMs. STAR [10], Traid-NVM [3], Anubis [26], cc-NVM [23], and Osiris [24] protect data confidentiality and integrity, but do not consider transaction-based failure-atomicity guarantee for a group of updates. SCA [13] and SuperMem [27] guarantee the crash consistency for data and its counter in transactions, but they do not consider data integrity, thus overlooking the expensive BMT updates. SCA proposes a selective counter atomicity scheme to reduce writes. However, for transactions that do not know the write set in advance [12], the memory barrier after each log write increases latency, thus decreasing the efficiency of SCA. SuperMem adopts a write-through counter cache to atomically persist the data and its counter by using a register. However, it becomes inefficient when extended to include the BMT integrity verification, since the register in SuperMem is occupied until the BMT root is updated, which stalls all subsequent memory writes.

In this paper, we propose Secon to efficiently guarantee the SEcurity and crash CONsistency for PM. There are three contributions behind Secon. First, Secon proposes **a scalable write-through security metadata cache** to atomically write data and its security metadata with high scalability. Based on our observation that in durable transactions there is always a consistent data version in PM (e.g., in log region or data region), Secon pre-persists the data and its security metadata from the register to PM without waiting for the BMT root to be updated. Hence, updating the BMT root is moved to the background, and the register is released early to allow the independent writes to be processed without waiting for a long time, thus improving the scalability. Second, to mitigate the ordering overheads in transactions for low latency, Secon proposes **a transaction-specific epoch persistency model** to only allow the order between the in-place write and its log. Third, Secon **colocates the counter with the log entry and coalesces the BMT blocks** to reduce the heavy write traffic caused by the security metadata, thus improving the PM endurance. Experimental results show that Secon respectively reduces the transaction latency and memory writes by 51% and 32% over the standard write-though scheme, and reduces the latency by 41% over SuperMem [27].

2 Background and Motivation

2.1 Threat Model

In general, the domain of a processor chip is considered to be secure [2, 6, 7, 13, 21, 25, 27], while the off-chip resources (e.g., memory bus and PM) can be attacked. An attacker can carry out *physical access* based attacks to obtain the data in PM via stealing the DIMM or snooping the memory bus to violate the confidentiality [2]. Moreover, the attacker can replay or tamper with the data in PM to break the integrity [26]. This paper aims to defend against these

[✉] Corresponding author: Yu Hua (csyhua@hust.edu.cn).

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530537

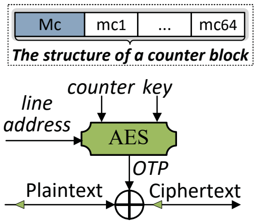


Figure 1. The process of counter-mode encryption.

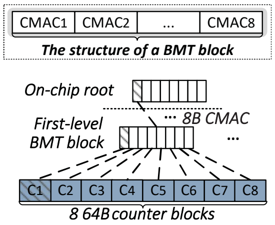


Figure 2. The structure of bonsai merkle tree.

attacks to protect the security (i.e., confidentiality and integrity) for the data stored in PM like prior studies [10, 19, 26].

2.2 Security Guarantee

Confidentiality Protection. Counter-mode encryption (CME) has been widely used to encrypt cachelines in the on-chip memory controller [2, 6, 7, 13, 21, 25, 27]. As shown in Fig. 1, an AES engine generates a one-time pad (OTP) to XOR with the plaintext cacheline to complete encryption. Hence, the cacheline becomes ciphertext when transmitted in the memory bus and stored in PM, thus protecting the confidentiality. When reading a cacheline from PM, CME decrypts the data by XORing the ciphertext with the OTP. The hot-spot counters are buffered in an on-chip counter cache, so the OTP generation can be overlapped with the ciphertext read to accelerate decryption. An OTP is generated using a private on-chip key, the physical line address, and a per-line counter block. A counter block (64B) contains one 64-bit major counter (Mc) and 64 7-bit minor counters (mc). Each mc corresponds to a cacheline. Hence, a counter block covers 64 cachelines in a 4KB page. For each memory write, the mc increases by 1 to generate a different OTP for encryption, which ensures that the OTP is never reused for a high security level. If a mc overflows, the Mc increases by 1 and all the mcs in this counter block are reset to 0 to re-encrypt the page for strong security guarantee [22].

Integrity Protection. Based on CME, the bonsai merkle tree (BMT) [19] verifies the integrity of the counter blocks. Fig. 2 shows the structure of BMT. A BMT block is 64B, which contains 8 CMACs (Counter Message Authentication Code). Each CMAC is 8B, which is computed by hashing (e.g., SHA-1) a lower-level 64B BMT block with an on-chip security key. Particularly, each CMAC in the first-level BMT block is computed using a 64B counter block. The counter blocks are leaf nodes in a BMT. Hence, one first-level BMT block cover 8 counter blocks, i.e., 8 pages. The root of BMT is stored in a non-volatile register on chip, which is not lost after a crash or power failure. For each memory write, after the counter block is updated, the entire BMT is updated from all the first level BMT blocks up to the root. The hot-spot BMT blocks are buffered in an on-chip BMT cache. When fetching a counter block from PM, this counter block is verified by calculating its parent BMT blocks layer by layer until finding a matched one in the BMT cache. In the worst case, this verification is processed up to the root.

2.3 Crash Consistency Guarantee

There are two requirements to ensure the crash consistency in the secure PM. First, the data itself needs to be atomically persisted. Since the atomic write is only 8B in 64-bit CPUs [11], the data larger than 8B are possible to be partially updated after a crash. To avoid this, write-ahead-logging (WAL) [15] in durable transactions is generally used to ensure the atomicity and durability for a group of writes. WAL first backs up old/new data in undo/redo logs, and then updates the in-place data. The write order between logs and data is guaranteed by using memory barriers (e.g., sfence). Even if a crash occurs when modifying the in-place data, the data can be recovered from logs to ensure consistency. For redo logging, the memory reads are redirected to the log region to identify the newest data. To avoid such redirections, our paper leverages undo logging in durable transactions to atomically persists a set of writes.

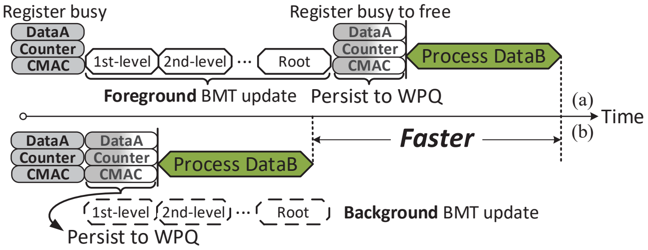


Figure 3. (a) The register is occupied for a long time. (b) The register is released as soon as possible to improve the scalability.

Second, the data and its security metadata (i.e., counters and BMT blocks) need to be atomically persisted for correct decryption and verification after a crash. Otherwise, we cannot trust the data. For example, if the data has been persisted to PM but its counter has not, the data cannot be correctly decrypted since the counter is inconsistent with the data. Moreover, if the data and the counter have been persisted but the BMT blocks have not, the data still cannot be correctly decrypted if the counter has been attacked.

2.4 Motivation

The Choices of Metadata Caches. To ensure crash consistency for secure PMs, there are two design choices for the security metadata cache to atomically persist the data and its security metadata, i.e., write-back [13] and write-through [27]. The write-back cache merges the updated metadata in cache, which reduces the metadata writes but requires new primitives to explicitly flush these metadata to PM [13], which makes it hard to port applications from unsecure PM to secure PM. To avoid this, we leverage the write-through cache that writes data along with its security metadata to PM.

Challenge. In the legacy write-through scheme [27], the long BMT update latency leads to low utilizations of the write pending queue (WPQ) and register in memory controller for a long time. As shown in Fig. 3a, though an in-flight write (e.g., DataA) is independent of the subsequent writes (e.g., DataB) in the WPQ, these subsequent writes are stalled until the register is released, which significantly decreases the throughput.

Insight. We observe that a transaction always maintains a *consistent copy of data* in the log region or data region. This observation brings our insight that the long-time occupation of the register or WPQ can be avoided. Specifically, the memory controller can persist the data and its metadata in advance, and release the register as soon as possible, leaving the BMT update to be executed in the background, as shown in Fig. 3b. Even if a crash occurs, the system can recover to a consistent state by using the correct logs or data. Based on this insight, we propose Secon to ensure security and crash consistency for PM with high scalability and performance.

3 The Secon Design

3.1 Overview

Fig. 4 illustrates the architecture of Secon. In the memory controller (MC), the write-through counter cache and BMT cache respectively store hot counters and BMT blocks. The write pending queue (WPQ)

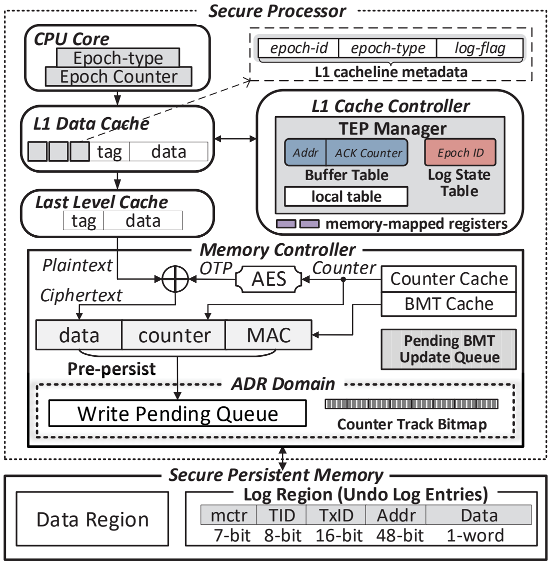


Figure 4. The architecture overview of Secon.

and a counter track bitmap are maintained in the battery-backed ADR domain [1], in which all the contents will be persisted to PM once a power failure or a crash occurs. In the L1 data cache controller, a Transaction-specific Epoch Persistency (TEP) manager relaxes the ordering constraints between undo logs and data. We rely on software schemes in the program such as locking [5] to guarantee the isolation between transactions that have conflicting accesses to the same data. When a cacheline is flushed to MC, the MC encrypts the cacheline and appends the tuple of <ciphertext, counter, first-level CMACs> (called *memory tuple*) to the WPQ. The tuples are coalesced in WPQ and atomically written to PM.

3.2 Scalable Write-through Metadata Cache

Secon designs a scalable write-through metadata cache to *pre-persist* the memory tuple to PM to remove the competitions of the register, thus enhancing the scalability. To ensure the atomicity of the data and its security metadata, we use a register in MC to temporarily store the memory tuple, i.e., <ciphertext, counter, first-level CMACs>. It is sufficient to only persist the first-level CMACs since we can rebuild a BMT by hashing the counters. Unlike SuperMem [27] that persists the memory tuple after updating the BMT root, Secon persists this tuple once it is stored in the register, thus moving the expensive BMT update to the background. In this way, all the independent writes can be served by MC once the register is released, which significantly improves the throughput.

Pre-persisting the memory tuple incurs a challenge that the new BMT root rebuilt after a system crash cannot match the on-chip old root even if no attack occurs. To tackle this challenge, Secon adds two structures in MC to index the updated metadata for recovery. First, the *pending BMT update queue* buffers the 48-bit physical addresses of CMACs that correspond to the write requests. Second, the *counter track bitmap* uses 64 bits as a unit, and each unit corresponds to a counter block. Each bit in the unit records which minor counter in the counter block is pre-persisted. The number of entries in the pending BMT update queue and the units in the counter track bitmap are both set to be 16, which consume 224B spaces. Note that for a 64-bit CMAC, using its 54 bits is sufficiently secure [20]. Secon hence leverages the unused 10 bits to record the ID of the counter block that is currently inconsistent with the on-chip BMT root. Fig. 5 shows the two structures. Supposing that the MC currently processes the data whose minor counter is mc1 and CMAC is CMAC1. ① After the counter and ciphertext are stored in the register, Secon stores the address of CMAC1 in the pending BMT update queue, searches the counter track bitmap for a free unit, and sets its first bit to 1 for mc1. ② The ID of this unit (i.e., 1) is stored in the unused bits of CMAC1. ③ After storing the first-level CMACs to the register, Secon adds the memory tuple to WPQ.

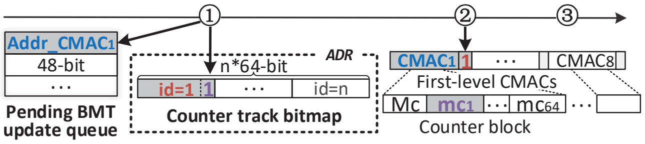


Figure 5. Guaranteeing the consistency between the off-chip counters and on-chip BMT root.

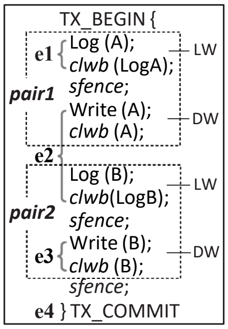
For background BMT update, Secon uses the CMACs (e.g., CMAC1) according to the addresses in the pending BMT update queue to update the BMT root. Existing BMT update schemes, such as streamlining update [6] and bonsai merkle forests [7], can be adapted in this process for fast BMT update. During the BMT update, Secon uses the ID stored in the unused bits of CMAC1 to read the 64-bit unit in the counter track bitmap. After updating the BMT, Secon resets the bits that are previously recorded in the unit. If the unit is equal to 0, Secon deletes the entry in the pending BMT update queue. Afterwards, the unused bits in CMAC1 are set to 0 in the BMT cache, and CMAC1 is written to PM.

In crash recovery, Secon traverses the first-level blocks of BMT to identify the CMACs whose unused bits are not 0. Since the counter track bitmap locates in ADR, it can provide the locations of the minor counters that have been pre-persisted while the BMT root has not been updated after crashes. Secon reads the 64-bit unit in the counter track bitmap based on the ID in the unused bits of a CMAC. According to the unit, Secon decreases the minor counters by 1 to keep the consistency between counters and the on-chip BMT root. Finally, the BMT is rebuilt by these counters.

3.3 Transaction-specific Epoch Persistency

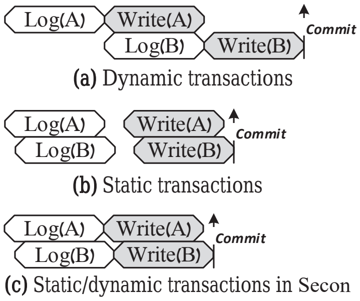
The background BMT update is efficient for independent writes. However, in transactions, since the data need to be updated after persisting its undo log to ensure atomicity, this ordering constraint incurs extra dependencies between writes, causing unnecessary BMT update latency in the critical path. For example, Fig. 6 shows a dynamic transaction, in which the write set is not predefined [12]. DataA and DataB do not have a dependency, and LogB is persisted to PM together with WriteA. However, since WriteA is ordered after LogA, LogB has to be delayed after LogA. Moreover, since WriteB is ordered after LogB, WriteB has to be delayed after WriteA, as shown in Fig. 7a. Therefore, LogB (or DataB) need to wait for the BMT updates of LogA (or DataA), thus causing unnecessary latency between persisting independent DataA and DataB.

To reduce ordering constraints, the epoch persistency model [17] divides a program into multiple epochs by memory barriers. All writes inside an epoch can be concurrently persisted. Only different epochs need to be persisted in order. For static transactions that predefine the write sets, epoch persistency works well since only one barrier is added between writing all the undo logs and new data, as shown in Fig.7b. However, for the widely used dynamic transactions without predefined write sets [12], a barrier is added after each log write, which weakens the efficiency of epoch persistency.



```
TX_BEGIN {
    Log (A);
    clwb (LogA);
    sfence;
    Write (A);
    clwb (A);
    Log (B);
    clwb(LogB);
    sfence;
    Write (B);
    clwb (B);
    sfence;
} TX_COMMIT
```

Figure 6. An example of using TEP.



(a) Dynamic transactions

(b) Static transactions

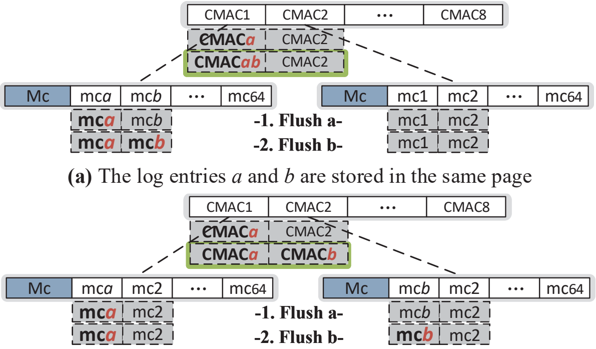
(c) Static/dynamic transactions in Secon

Figure 7. The timeline of executing transactions.

To minimize the ordering constraints in both static and dynamic transactions, Secon proposes a Transaction-specific Epoch Persistency (TEP) model. TEP extends the epoch persistency model by supporting *paired epoch*, i.e., two adjacent epochs are paired. The writes in the same pair are persisted as the epoch order, but different pairs are concurrently persisted. A regular epoch needs to be persisted after the previous paired and regular epochs. Fig. 6 shows an example of using TEP in a dynamic transaction, in which LW (DW) stands for the log write (data write). The epochs of $e1$, $e2$, $e3$ are divided into two pairs, in which the DW and LW in $e2$ are respectively paired with $e1$ and $e3$. The transaction commit is a regular epoch ($e4$). By using TEP, the two pairs are simultaneously persisted, as shown in Fig. 7c. As such, it is unnecessary for `LogB` (or `DataB`) to wait for the BMT updates of `LogA` (or `DataA`), thus removing the unnecessary ordering constraints.

Secon leverages software interfaces and hardware extensions to implement TEP. We provide a *set_log_region (start, end)* interface to initialize a log region in PM. Two memory-mapped registers in L1 cache controller record the *start* and *end* addresses of the log region, as shown in Fig. 4. Hence, Secon easily identifies a log write to the log region. In the CPU core, each hardware thread maintains a 1-bit epoch-type and an 8-bit epoch counter. The epoch-type is set to "1" on log writes (i.e., paired epoch) and "0" on transaction commits (i.e., regular epoch). The epoch counter increases on each memory barrier as the current epoch ID. In the L1 data cache (L1D), three fields are added to each cacheline: (1) The 8-bit epoch-id stores the epoch of the write. (2) The 1-bit epoch-type is the same as that in hardware thread. (3) The 1-bit log-flag denotes whether the write is a log write ("1" means yes). Using an 8-bit epoch-id is sufficient since most transactions have 5–50 epochs in practice [16]. If the epoch-id overflows, Secon persists all transaction writes in L1D, and resets the next epoch-id to 0.

Secon adds a TEP manager in the L1 cache controller to control the write order. Inside the TEP manager, a buffer table stores: (1) The 48-bit physical address (Addr) of the data cacheline waiting for its dependent cachelines to persist. (2) An 8-bit ACK counter that increases upon persisting a cacheline and decreases upon receiving an ACK from the memory controller (MC). Each <Addr, ACK Counter> entry belongs to a hardware thread. The local table stores the physical addresses of cachelines waiting to be persisted. To handle the case that a log cacheline is evicted to MC but its BMT root has not been updated, we use a log state table to store the epoch-ids of evicted log cachelines to avoid that the data is persisted before updating the BMT root of the data's log. Once the ACK of the log cacheline is received from MC, the epoch-id entry is deleted. Each epoch-id entry belongs to a hardware thread. In summary, TEP only consumes less than 1KB space when using 16 hardware threads, a 32KB L1D, and a 40-entry local table.



(a) The log entries a and b are stored in the same page

(b) The log entries a and b are stored in different pages

Figure 8. The contents of the counter and BMT blocks when flushing the log entries a and b.

During transaction processing, when persisting a data cacheline, e.g., d, Secon creates an entry in the buffer table, stores the physical line address in its Addr field, and persists the cachelines that d depends on. There are two cases to persist the dependent cachelines:

Case1: The epoch-type of d is paired epoch. d is persisted only after persisting its corresponding log cacheline. Supposing that the epoch-id of d is x. The TEP manager simultaneously searches: (1) The log state table for an entry whose epoch-id is equal to $x - 1$. (2) The L1D for a cacheline (e.g., c) whose epoch-id is equal to $x - 1$ and log-flag is equal to "1". If an entry is found in the log state table, Secon persists d after releasing this entry. Otherwise, Secon persists c before d. The search process is very fast since all entries in the log state table and all the L1D cachelines are checked in parallel, which only consumes several cycles for tag matching.

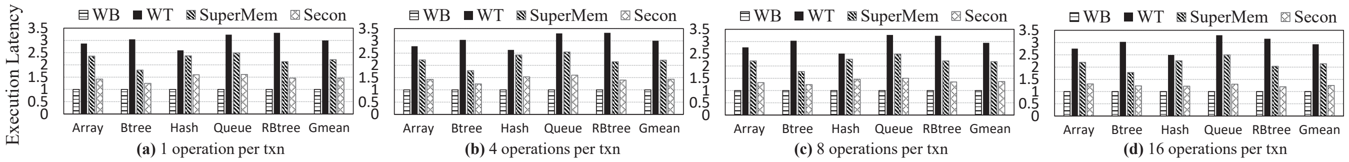
Case2: The epoch-type of d is regular epoch. Secon persists all cachelines whose epoch-ids are smaller than x. Once the ACK counter is equal to 0, d is persisted.

3.4 Write Reduction for Security Metadata

The write-through approach is efficient to atomically persist data and its security metadata for each write, which however increases the write traffic. To reduce the metadata writes, Secon proposes two schemes as follows to reduce the counter and BMT writes.

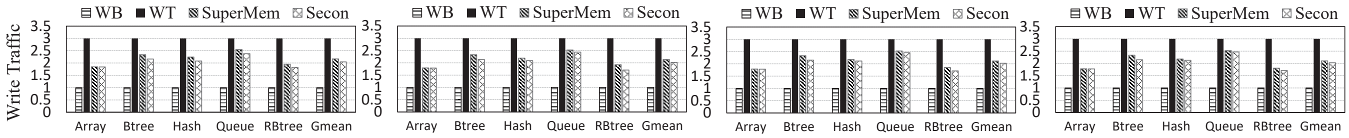
3.4.1 Co-locate Log and Counter. In Secon, the undo log entry contains an 8-bit thread ID (TID), a 16-bit transaction ID (TxID), a 48-bit data address (Addr), and an 8B word to record the old data. The size of an undo log entry is 18B. For each cacheline write, only the 7-bit minor-counter is updated if the minor-counter does not overflow. Hence, Secon co-locates the minor counter with the log entry in one cacheline to avoid consuming extra cachelines to write counters, thus reducing the write traffic. To achieve this, we add a "mctr" field in the log entry to store the minor-counter, as shown in the "Log Region" in Fig. 4. If the write is a log write, the MC adds the minor-counter to the log entry before persisting the memory tuple. If the minor-counter overflows, Secon writes the 64B counter block to ensure that the counter in PM is at the newest state.

3.4.2 Coalesce BMT Blocks. We explore and exploit the spatial locality of BMT blocks to reduce the write traffic of BMT. As mentioned in § 2.2, the first-level BMT blocks cover 8 pages. Hence, the data in these 8 pages will persist the same BMT block to PM, causing high write redundancy. As logs are stored in a continuous



(a) 1 operation per txn **(b)** 4 operations per txn **(c)** 8 operations per txn **(d)** 16 operations per txn

Figure 9. The transaction (txn) execution latencies with different numbers of operations per transaction (normalized to WB).



(a) 1 operation per txn **(b)** 4 operations per txn **(c)** 8 operations per txn **(d)** 16 operations per txn

Figure 10. The numbers of memory writes with different numbers of operations per transaction (normalized to WB).

log region with good spatial locality, the same BMT block will be frequently persisted when flushing log entries. We observe that regardless of whether the data are stored in the same page or in different pages (within 8 pages), the latter BMT block updates always include the contents of the former BMT block updates. Specifically, as shown in Fig. 8, when processing log entries *a* and *b* in an arbitrary order, the BMT block finally contains the updated CMACs of *a* and *b* when the two logs are stored in the same page or different pages. Based on this observation, Secon coalesces the BMT blocks: once a BMT block is added to the WPQ, Secon searches the WPQ for a BMT block whose physical address is the same as the coming one. If found, Secon deletes the former BMT block since the latter one already contains all the updates, thus reducing the BMT writes.

4 Performance Evaluation

4.1 Methodology

We use the cycle-accurate Gem5 simulator [4] with NVMain [18] to implement and evaluate Secon. We model Gem5 to simulate a multi-core PM system. Table 1 shows the system configurations.

We use five micro-benchmarks and two macro-benchmarks for evaluation. The micro-benchmarks are widely used in existing NVM systems [9, 13, 27]. The value sizes are set to 256B:

- **Array.** Swap two random entries in a 1GB array.
- **Queue.** Enqueue/dequeue random entries in a 1GB queue.
- **Btree.** Insert/delete random nodes in a 1GB B-tree.
- **Hash.** Insert/delete random items in a 1GB hash table.
- **RBtree.** Insert/delete random nodes in a 1GB red-black tree.

The macro-benchmarks are adopted from WHISPER [16]:

- **YCSB.** Cloud benchmark. 100% update.
- **TPCC.** OLTP benchmark. Using the New-Order transaction.

We compare our proposed Secon with the following designs:

- **An ideal write-back scheme (WB).** WB employs a large battery backed write-back cache to store security metadata without the overheads of BMT updates and extra metadata writes.
- **A write-through scheme (WT).** WT employs a write-through metadata cache, in which each memory write and security metadata are atomically written to memory after the BMT root is updated.
- **The state-of-the-art design (SuperMem [27]).** Our BMT coalescing scheme is applied to SuperMem to ensure data integrity.

4.2 Transaction Latency and Throughput

4.2.1 Single-core Performance. Fig. 9 shows the execution latencies of micro-benchmarks. The results are normalized to WB. WT shows 3× higher latency than WB, since WT triples the memory writes and puts the expensive BMT update on the critical path of transaction execution. Compared with WT, SuperMem reduces the execution latency by 26% due to using its metadata coalescing and cross-bank writes. Secon improves performance by 41% over SuperMem on average, due to using our scalable metadata cache to write the memory tuple in advance to accelerate the processing of independent write requests. To demonstrate the efficiency of TEP, we modify the micro-benchmarks to allow one transaction to contain 1–16 operations to increase the transaction size and memory barriers. The results show that Secon further reduces the transaction execution latency by 14% on average due to using paired epochs to mitigate ordering constraints. For example, the execution latency of Hash with 16 operations decreases by 24% compared with the 1-operation configuration.

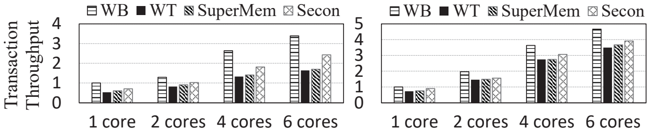
Table 1. Configurations of the simulation system.

Processor	
CPU	6 cores, X86-64, out-of-order, 2 GHz
L1 Cache	Private, 64KB, 8-way, 2 cycles
L2 Cache	Private, 256KB, 8-way, 12 cycles
LLC	Shared, 8MB, 8-way, 30 cycles
Counter Cache	256KB, 8-way, LRU, 8 cycles [27]
BMT Cache	256KB, 8-way, LRU, 8 cycles
Memory Controller	FRFCFS, 32 entries of WPQ
Backend Operations	En/decryption and BMT hash: 40ns [14]
Persistent Memory	
Capacity	16GB Phase-change memory (PCM)
Latency Model	tRCD/tCL/tCWD/tFAW/tWTR/tWR = 48/15/13/50/7.5/300 ns [27]

4.2.2 Multi-core Performance. Fig. 11 shows the transaction throughput on macro-benchmarks using different numbers of CPU cores. The results are normalized to WB using 1 core. Compared with SuperMem, Secon improves the throughput by 19%/43% on TPCC/YCSB, since Secon pre-persists the memory tuples to improve the scalability of the metadata cache.

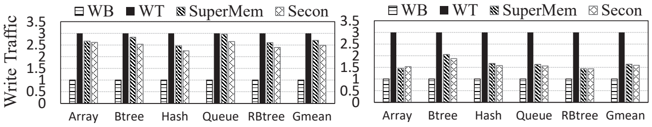
4.3 Write Traffic

Fig. 10 shows the numbers of writes to PM on micro-benchmarks, which are normalized to WB. WT incurs 3× writes than WB due to writing ciphertext, counter, and the first-level BMT blocks for each write. Secon reduces the writes by 32% over WT due to coalescing the counter and BMT writes. SuperMem also merges metadata to reduce memory writes. When the number of operations in a transaction increases from 1 to 16, Secon and SuperMem slightly decrease the write traffic. Fig. 12 shows that when increasing the value size from 64B to 1024B in the micro-benchmarks, Secon reduces the memory writes by 17% to 47% than WT since the increase of the spatial locality reduces the amount of metadata writes.



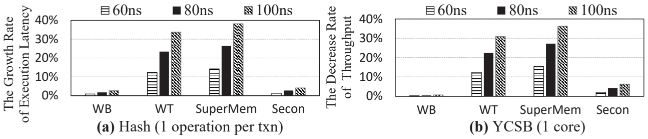
(a) YCSB (b) TPCC

Figure 11. The normalized transaction throughput of macro-benchmarks using different numbers of CPU cores (normalized to WB-1core).



(a) 64B value size (b) 1024B value size

Figure 12. The normalized write traffic in different value sizes (1 operation per transaction, normalized to WB).



(a) Hash (1 operation per txn) (b) YCSB (1 core)

Figure 13. The effects of BMT hashing latency on transaction latency and throughput.

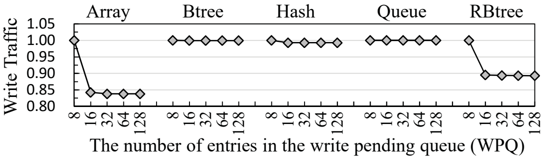


Figure 14. The normalized write traffic in different WPQ sizes.

4.4 Sensitivity Study

4.4.1 BMT Hashing Latency. We study how the hashing latency in BMT update impacts the transaction performance. Prior designs set the hashing latency to different numbers, such as 40ns [14] and 80ns [8]. We vary the hashing latencies to {40, 60, 80, 100}ns, and present the results on Hash and YCSB (other results are not shown here due to the limited space). Fig. 13 shows the growth rate of execution latency and the decrease rate of throughput compared with the 40ns latency. When increasing the hashing latency, Secon only increases the execution latency by less than 5%, and decreases the throughput by less than 7%, due to writing the memory tuples in advance and moving the expensive BMT update to the background. Therefore, Secon is not sensitive to the hashing latency. However, WT and SuperMem are sensitive to the hashing latency due to waiting for the update of BMT root. When setting the hashing latency to 100ns, they respectively slow down the execution by 34% and 38%, and decrease the throughput by 31% and 36%.

4.4.2 Write Pending Queue Size. We study how the size (i.e., the number of entries) of the write pending queue (WPQ) affects the write traffic by changing the WPQ size from 8 to 128 entries. The results are normalized to the 8-entry WPQ. Fig. 14 shows that when increasing the size from 8 to 32 entries, the write traffic of Array and RBtree sharply decrease since they have good localities to coalesce many BMT blocks in the WPQ. From the experimental results, we learn that reserving a 32-entry WPQ is sufficient to reduce the metadata writes, while not heavily increasing the hardware overhead of WPQ in the memory controller.

5 Conclusion

This paper proposes Secon to efficiently bridge the gap between crash consistency and security for persistent memory systems. Secon leverages a scalable write-through security metadata cache to atomically pre-persist the data and its security metadata, thus eliminating unnecessary write stalls in the memory controller. Moreover, Secon leverages a transaction-specific epoch persistency model to mitigate the ordering constraints in transactions. Secon further co-locates the logs with counters and coalesces BMT blocks to reduce the metadata writes. Experimental results demonstrate that Secon outperforms the state-of-the-art schemes on transaction latency, throughput, and write traffic.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 62125202 and Key Laboratory of Information Storage System, Ministry of Education of China.

References

[1] 2016. Deprecating the PCOMMIT Instruction. https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html.

[2] A. Awad, P. J. Manadhata, S. Haber, Y. Solihin, and W. G. Horne. 2016. Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers. In *ASPLOS*.

[3] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair. 2019. Triad-NVM: persistency for integrity-protected and encrypted non-volatile memories. In *ISCA*.

[4] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011).

[5] D. R. Chakrabarti, H. Boehm, and K. Bhandari. 2014. Atlas: leveraging locks for non-volatile memory consistency. In *OOPSLA*.

[6] A. Freij, S. Yuan, H. Zhou, and Y. Solihin. 2020. Persist Level Parallelism: Streamlining Integrity Tree Updates for Secure Persistent Memory. In *MICRO*.

[7] A. Freij, H. Zhou, and Y. Solihin. 2021. Bonsai Merkle Forests: Efficiently Achieving Crash Consistency in Secure Persistent Memory. In *MICRO*.

[8] B. Gassend, G. Edward Suh, D. E. Clarke, M. V. Dijk, and S. Devadas. 2003. Caches and Hash Trees for Efficient Memory Integrity Verification. In *HPCA*.

[9] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *ISCA*.

[10] J. Huang and Y. Hua. 2021. A Write-Friendly and Fast-Recovery Scheme for Security Metadata in Non-Volatile Memories. In *HPCA*.

[11] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. 2019. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *SOSP*.

[12] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *ASPLOS*.

[13] S. Liu, A. Kolli, J. Ren, and S. M. Khan. 2018. Crash Consistency in Encrypted Non-volatile Main Memory Systems. In *HPCA*.

[14] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan. 2019. Janus: optimizing memory and storage support for non-volatile memory systems. In *ISCA*.

[15] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992).

[16] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *ASPLOS*.

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

[18] M. Poremba, T. Zhang, and Y. Xie. 2015. NVMain 2.0: A User-Friendly Memory Simulator to Model (Non-)Volatile Memory Systems. *IEEE Comput. Archit. Lett.* 14, 2 (2015).

[19] J. Rakshit and K. Mohanram. 2017. ASSURE: Authentication Scheme for SecURE Energy Efficient Non-Volatile Memories. In *DAC*.

[20] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi. 2018. Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories. In *MICRO*.

[21] S. Swami, J. Rakshit, and K. Mohanram. 2016. SECRET: smartly EnCRypted energy efficient non-volatile memories. In *DAC*.

[22] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *ISCA*.

[23] F. Yang, Y. Lu, Y. Chen, H. Mao, and J. Shu. 2019. No Compromises: Secure NVM with Crash Consistency, Write-Efficiency and High-Performance. In *DAC*.

[24] M. Ye, C. Hughes, and A. Awad. 2018. Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories. In *MICRO*.

[25] V. Young, P. J. Nair, and M. K. Qureshi. 2015. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *ASPLOS*.

[26] K. A. Zubair and A. Awad. 2019. Anubis: ultra-low overhead and recovery time for secure non-volatile memories. In *ISCA*.

[27] P. Zuo, Y. Hua, and Y. Xie. 2019. SuperMem: Enabling Application-transparent Secure Persistent Memory with Low Overheads. In *MICRO*.