Silo: Speculative Hardware Logging for Atomic Durability in Persistent Memory

Ming Zhang and Yu Hua*
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
Corresponding Author: Yu Hua (csyhua@hust.edu.cn)

2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10071034

***Abstract*—Persistent memory (PM) provides data persistency. Due to this property, guaranteeing atomic durability becomes important for applications running on PM in order to ensure the crash consistency for a group of updates. To this end, hardware logging has received many attentions by overlapping the log operations and transaction execution. Unfortunately, existing approaches regard logs as backups, which inevitably increases the log writes to PM, thus exacerbating the limited endurance of PM and imposing constraints on the write ordering.**

This paper proposes Silo, a speculative hardware logging design to ensure atomic durability with ultra-low overheads. Unlike existing studies, Silo exploits a speculative methodology and regards logs as data to make the common case fast. In practice, system crashes or power failures rarely occur for a machine. Hence, we do not need to write logs to back up data in most of the running time. Based on this observation, Silo temporarily maintains the undo+redo logs on chip during transaction execution. After the transaction commits, Silo leverages the new data in these on-chip logs to in-place update the PM data region, instead of conservatively writing logs as useless backups in common cases where no crash occurs. In this way, Silo significantly reduces the write overheads. If a crash occurs, Silo still efficiently flushes these on-chip logs to PM for recovery without any loss of correctness. Experimental results demonstrate that Silo significantly improves the transaction throughput by 4.3×, and reduces the memory writes by 76.5% compared with state-of-the-art designs.

I. Introduction

Persistent memory (PM) [3], [7], [23], [53] provides salient properties, such as non-volatility, high density, and DRAM-like performance. By using PM, applications can directly access the persistent data using `load` and `store` instructions without serializing data to the file systems or invoking expensive system calls. Nevertheless, a fundamental requirement needs to be satisfied when using PM, i.e., guaranteeing the atomic durability. Due to non-volatility, the data in PM are not lost even if a system crash or power failure occurs. As a result, the persistent data could be partially updated to cause inconsistency. To provide data consistency, we need to guarantee the atomic durability, which requires that either *all* or *none* of the updates are applied to PM in case of a crash.

To ensure atomic durability, hardware logging has received many attentions [2], [16], [25], [27], [28], [38], [46], [52]. These approaches offload log operations to hardware to overlap logging with transaction execution, as long as the logs are persisted before the updated data to support crash recovery. However, designing an efficient hardware logging scheme is non-trivial. Due to conservatively writing logs to the PM log region in each transaction, legacy hardware logging schemes incur high overheads in two aspects, as presented below.

1) Heavy Writes. Most of existing hardware logging approaches consume 2–3× memory writes by additionally backing up the data in the log entries. In particular, the undo, redo, and undo+redo log entries respectively record the old, new, and old+new data. Writing logs supports crash recovery, but the write traffic significantly increases, which exacerbates the write endurance of PM and hence shortens the PM lifetime.

2) Ordering Constraints. When writing logs and updated data to PM, prior hardware logging studies suffer from ordering constraints. Specifically, the undo loggings [28], [46] wait for persisting all the updated data before the transaction commits. In the redo loggings [16], [25], [27], the new data cannot be in-place updated until all the logs are persisted. For undo+redo loggings, FWB [38] forces logs to PM before the updated data for each write. MorLog [52] waits for flushing all logs in the L1 cache and log buffers to PM before commit to guarantee durability. Such ordering constraints increase the latency and lead to performance degradation.

Existing studies do not efficiently address the above challenges. To reduce log writes, ATOM [28] packs the log metadata into one cacheline, ReDU [25] supports log coalescing, and MorLog [52] eliminates unnecessary redo logs. However, these schemes still need to write logs to PM in each transaction. Proteus [46] adopts a log pending queue to discard undo logs, but the transaction commit needs to wait for flushing the updated cachelines, and the last log entry in each transaction is flushed to indicate the commit. ASAP [2] relaxes the durability requirement to asynchronously persist the undo logs and data after commit, but it needs to record the data and control dependencies in hardware. Moreover, the logs still cause extra PM writes.

In practice, a single machine normally runs without crash or power failure in most of the time [6], [18]. We refer to this situation as common failure-free case. We observe that in such case all the logs written to PM are truncated after the transaction commits. Although these logs are not used, they incur high write overheads. Furthermore, we observe that it is unnecessary to write logs in common failure-free cases, since the data in PM will not be partially updated. Instead, we can write logs in a *speculative* manner, i.e., only writing logs to PM upon crashes for recovery. In this way, we avoid

978-1-6654-7652-2/23/$31.00 ©2023 IEEE

the logging overheads in common failure-free cases, but also guarantee the recoverability even if a crash occurs.

Based on the above observations, we propose Silo, a speculative hardware logging design to ensure atomic durability for PM. Unlike prior studies that conservatively write logs to the PM log region for backing up data, Silo leverages a novel idea of "Log as Data", which utilizes the on-chip logs to directly in-place update the PM data region in common failure-free cases. Only in rare cases (e.g., crashes), Silo flushes the on-chip logs to the PM log region for recovery. Such a speculative logging makes the common case fast.

Silo maintains the undo+redo logs from one transaction in a small log buffer in memory controller. These logs are merged to reduce the space overhead of the log buffer. We observe that the new data recorded in logs already contain all the updates of a transaction. Hence, in common failure-free cases, Silo flushes these new data to in-place update the data region after commit, thus ensuring durability. Other contents recorded in logs are simply discarded on chip. The new data are further coalesced in an internal buffer in PM DIMM to reduce the write amplification to the physical media. Due to not writing log entries to back up data, Silo efficiently *mitigates the write traffic* caused by log writes. Moreover, since Silo exploits the logs for in-place updates, the transaction commit does not need to wait for writing logs (and modified cachelines) to the PM log (and data) regions, thus *removing the ordering constraints*.

Apart from making the common case fast, Silo also guarantees the correctness in two rare cases. First, if the logs overflow from the on-chip log buffer, Silo flushes the overflowed undo logs to the PM log region in a batch manner. These undo logs ensure the atomicity. Flushing the overflowed logs and adding new logs are performed in parallel to reduce the overall latency. Second, if a crash occurs, Silo selectively flushes the necessary on-chip logs to PM log region for correct recovery. Specifically, if the crash occurs before commit, Silo flushes the undo logs to revoke the partial updates to ensure atomicity. However, if the crash occurs after commit, Silo flushes the redo logs to replay all the transaction updates to ensure durability. Therefore, Silo still guarantees atomic durability in rare cases. To support the log flushing upon a crash, we implement the log buffer to be persistent by using a small battery [5].

In summary, this paper makes the following contributions:

- We propose Silo, a hardware logging design that makes the common case fast to mitigate the write traffic and ordering constraints with the atomic durability guarantee.
- We employ the new data recorded in on-chip logs to in-place update the PM data region in common failure-free cases, instead of conservatively writing these logs to the PM log region as useless data backups.
- We efficiently handle rare cases including log overflow and system crashes by only flushing necessary logs to PM without compromising the correctness and recoverability.
- We conduct extensive experiments. The results demonstrate that Silo improves the transaction throughput by 4.3×, and reduces 76.5% of memory writes over the state-of-the-art design [52].

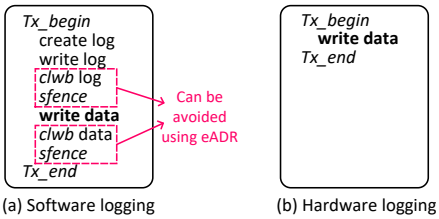

Fig. 1. The example codes of (a) software logging, and (b) hardware logging.

II. Background and Motivation

A. Atomic Durability for PM

Due to offering data persistency in PM, it is critical to ensure *atomic durability* for crash consistency. *Atomicity* requires that a group of updates are applied to PM in an all-or-nothing manner in case of crashes, and *durability* requires that these updates must be persisted in PM after commit [10]. Commodity hardware has provided instructions for atomic update. However, the granularity of atomic CPU write is only 8B in 64-bit CPUs [32], and the atomic instructions (e.g., `CAS`) in concurrent programming are used for single writes.

To overcome these limitations, existing PM systems employ the concept of *transaction* from the database community to ensure atomic durability for a group of updates [25], [52]. A transaction wraps multiple reads and writes together by using the pair of `Tx_begin` and `Tx_end` interfaces. A transaction must succeed or fail as a complete unit. Due to offering simple programming model and strong guarantees, the transaction becomes an important building block for PM applications, which has been widely recognized in both industry [24], [39] and academia [14], [25], [27], [34], [38], [48], [52], [56].

B. Why Hardware Logging

To implement the atomic durability in transactions, write-ahead logging (WAL) [36] is widely employed. WAL first backs up data in persistent logs, and then allows in-place updates. Hence, even if a crash occurs in the middle of a transaction, we can use logs to recover the corrupted data. WAL perfectly meshes with PM since logging works at a fine granularity (e.g., word) for a set of scattered writes [48], which fits the byte-addressability property of PM.

WAL can be performed in software or hardware. As shown in Fig. 1a, software logging needs to persist logs (e.g., using `clwb`) before the corresponding data. The durability order is controlled by memory barriers (e.g., using `sfence`). All log operations exist on the critical path, which decreases the transaction throughput by up to 70% [28]. Different from software logging, Fig. 1b shows that hardware logging only needs to annotate the transaction boundaries using the `Tx_begin` and `Tx_end` interfaces for general-purpose use. All log operations and data persistence are performed by hardware in the background. By overlapping log operations with transaction execution, hardware logging significantly improves the performance over software logging schemes [38].

C. Using eADR for Software Logging is Expensive

Recently, Intel proposes eADR [22] that uses a large battery to make the entire CPU cache become persistent. Hence, by

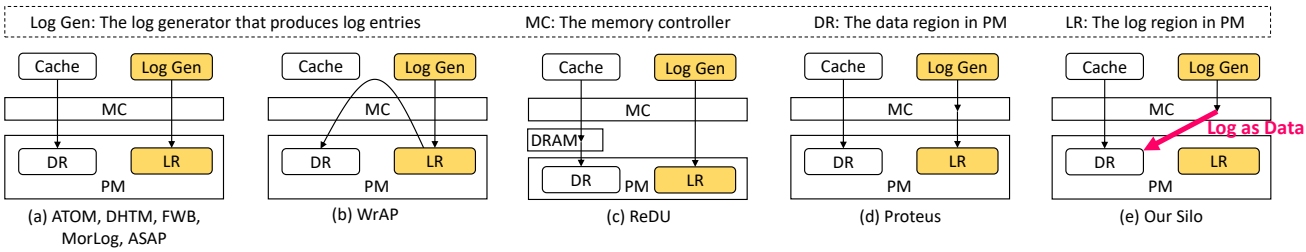

Fig. 2. The comparisons of the logging designs among existing hardware logging studies (a–d) and our Silo (e).

using eADR, the `clwb` and even `sfence` in Fig. 1a can be avoided in the Intel x86 TSO model [20], [45]. However, it is non-trivial to use eADR for software logging to support atomic durability due to two reasons. First, to support atomicity, the programmers need to write logs to the CPU cache to back up data, which unfortunately increases cache writes and pollutes the cache. Specifically, the logs are generated in an append manner, and hence have different physical addresses. Therefore, these logs cannot be merged in cache to reduce the number of logs, even if they record the same data's updates. Note that in common failure-free cases, the substantial cached logs are useless due to no need of recovery. However, these logs frequently write the CPU cache and cause random data evictions, which incurs heavy cache accesses and exacerbates the application locality, leading to low performance [8].

Second, to support durability, eADR requires a large-capacity battery for the entire cache, which significantly increases the energy consumption, hardware cost, and system maintenance burdens, as reported by existing works [5], [15], [57]. Hence, the ADR platform is still in its lifetime, and both ADR and eADR platforms will co-exist to support PM systems, as pointed out by Intel [45]. In line with recent studies on PM [2], [13], [15], [57], our work is based on the ADR platform, in which the CPU cache is volatile.

Given the above considerations, we leverage hardware logging, instead of relying on eADR, to support atomic durability. Hardware logging can control the ways to store and write logs without the overheads of cache pollutions and large batteries.

D. Ordering Constraints in Hardware Logging

Different types of hardware loggings exhibit various ordering constraints, as shown in Fig. 3.

Undo. The hardware undo logging allows the updated data (e.g., `Data A`) to be persisted after the log (e.g., `Ulog A`). As the old data are backed up in undo logs, the system can recover by revoking the partial updates after crashes. But the transaction commit needs to wait for persisting all the updated data to guarantee durability. Otherwise, the new values are lost in case of a crash.

Redo. The hardware redo logging allows the transaction to commit after all the redo logs have been persisted. As the new data are backed up in redo logs, the system can recover by replaying these updates. However, to guarantee atomicity, the in-place data cannot be updated until all the redo logs are persisted. Otherwise, the in-place old data are partially overwritten if a crash occurs in the middle of the transaction.

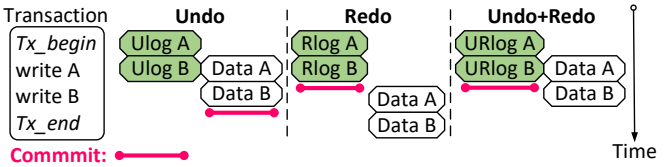

Fig. 3. The comparisons of the persist orderings among hardware undo, redo, and undo+redo loggings.

Undo+Redo. By backing up old+new data, the hardware undo+redo logging enables the data to persist after logs, and allows the transaction to commit without waiting for persisting all the updated data. Nevertheless, legacy undo+redo designs still suffer from extra ordering overheads. Specifically, FWB [38] forces the logs to PM before the updated data for each write. MorLog [52] waits for flushing all the logs in L1 cache and log buffer before the transaction commits to guarantee durability. Moreover, due to backing up old+new data, the undo+redo logging increases the write traffic to PM.

E. Log As Data

Existing hardware logging studies adopt different logging designs. Fig. 2a–d outline each of these designs. (a) ATOM [28], DHTM [27], FWB [38], MorLog [52], and ASAP [2] write the logs to the log region to back up data, and write the modified cachelines to the data region to install the transaction updates. (b) WrAP [16] writes redo logs to the log region and reads these logs to update the data region, thus causing extra reads. (c) To avoid such reads, ReDU [25] uses DRAM to buffer the modified cachelines. After commit, these cachelines directly update the data region. The redo logs are still written to the log region. (d) Proteus [46] buffers the undo logs on chip and these logs (except the last one) are discarded after commit. However, the transaction commit needs to wait for writing the modified cachelines to the data region. Such ordering constraints decrease the performance.

In general, existing designs follow the traditional "Log as Backup" methodology to conservatively write logs to PM as data backups for crash recovery. However, since in practice the crash rarely occurs for a single machine [6], [18], the logs are written in vain, which increases write traffic and imposes ordering constraints, as analyzed in § I.

Unlike prior studies, our proposed Silo leverages a novel "Log as Data" idea to use the new data in on-chip logs to update the PM data region after commit in common cases (i.e., no crash occurs), as shown in Fig. 2e. In this way, it is unnecessary to write logs as data backups, thus reducing the write traffic. Moreover, the transaction commit does not

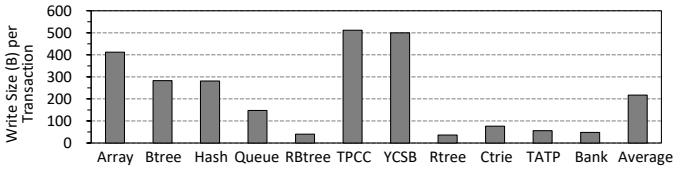

Fig. 4. The write size (in bytes) in one transaction.

need to wait for: 1) flushing the modified cachelines to the data region, and 2) flushing the logs to the log region, thus mitigating the ordering constraints. If a crash occurs, Silo flushes the logs to log region for recovery. In such a speculative logging manner, Silo significantly reduces the overheads to accelerate the transaction commit in common cases.

To support the "Log as Data" idea, we store the logs in an on-chip log buffer during transaction execution. After the transaction commits, these logs are flushed to the data region for in-place updates. Furthermore, to support the functionality of flushing logs on a crash, we implement the log buffer as a persistent buffer, which is backed by a small battery. This implementation is compatible with existing works that require on-chip persistent buffers for different purposes, e.g., storing metadata [28] and tracking dependencies [2].

The log buffer needs to be small due to the limited on-chip space. We leverage the applications in Table III, the workloads from PMDK [24] (i.e., Radix tree or Rtree and Crit-bit trie or Ctrie that perform insert operations), a telecom application transaction processing (TATP) benchmark [21], and a banking application [4], to evaluate the write size in one transaction. As shown in Fig. 4, the write size is generally less than 0.5 KB per transaction. The transactions in many real-world PM applications modify small amounts of data (referred as *small write set* in this paper). There are two reasons behind this observation: 1) A small write set reduces the overheads of enforcing persistency and resolving write conflicts. Hence, the online transaction processing (OLTP) applications typically involve a small write set [31], [40], [60]. Note that we do not care about the size of the read set in transactions, since the read operations do not produce logs. 2) Transactions are also used to wrap the critical code regions for concurrency control (e.g., in HTM [27]). To achieve high concurrency, the critical code region is small to efficiently mitigate blocking. Based on the above reasons, using a small log buffer is sufficient to maintain the logs in one transaction, since small write sets commonly exist. Even if the logs overflow from the buffer in some large transactions, we efficiently handle this issue by flushing undo logs without any transaction abort (§ III-F).

III. The Silo Design

A. Assumption

Like existing studies [25], [28], [38], [46], [52], we assume that the atomic durability is based on hardware logging in the ACID transactions. The isolation between conflicting transactions is supported by software mechanisms in programs, such as fine-grained locking [11]. Moreover, we currently do not support nested transactions since they are orthogonal to atomic durability [25], [52].

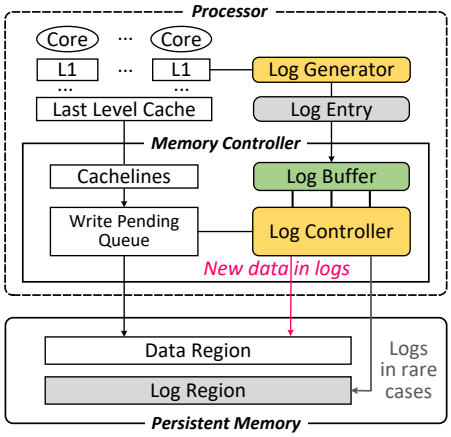

Fig. 5. The architecture of Silo.

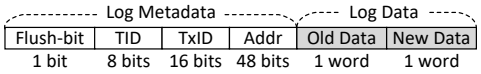

Fig. 6. The structure of the log entry.

B. Architecture

Fig. 5 illustrates the architecture of Silo. The key hardware components are presented as follows.

Log Entry. We design the undo+redo log entry to record the log metadata and log data (including the old and new data), as shown in Fig. 6. Log entries are generated and controlled by hardware. The 8-bit *tid* and 16-bit *txid* respectively record the thread ID and transaction ID [38], [52]. The 48-bit *addr* records the physical address of the log data. The *flush-bit* avoids unnecessary writes, as presented in § III-D. The size of the old or new data is 1 word (e.g., 8B in 64-bit CPUs) to record the data change made by a CPU `store` instruction.

Log Generator. Silo designs a log generator in each L1 data cache (L1D) controller to generate log entries when the L1D is updated during a transaction. When `Tx_begin` is executed, the log generator records the current tid, and increases the value stored in a specific register as the txid [52]. When a cacheline is modified, the log generator captures the new data and its physical address from the in-flight write request. The old data is simultaneously obtained from L1D. Reading the old data does not incur extra latency since it is overlapped with the tag matching of the in-flight write [33]. After obtaining the log data, the log generator encapsulates them with tid, txid, and log data's address into a log entry (the flush-bit is 0), and assigns a physical address for this log entry in the PM log region. Afterwards, the log generator sends the log entry to a log buffer in the memory controller. The CPU `store` completes without waiting for sending the log entry, since sending log entries is independent on the next instruction execution. When `Tx_end` is executed, the log generator stops producing logs.

Log Buffer. Each CPU core has a small private battery-backed log buffer to maintain the log entries from one transaction. Each core's log buffer contains 20 entries based on the results in § VI-D. Beside each entry, a 64-bit hardware comparator [1] is used for fast address comparisons. The logs are written to the log buffer in order. After a transaction commits, the entries in log buffer are deallocated to serve the next transaction.

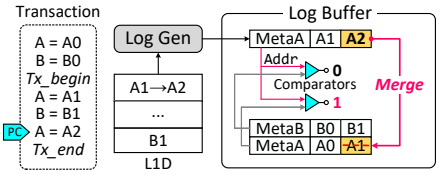

Fig. 7. The logs are merged in the log buffer.

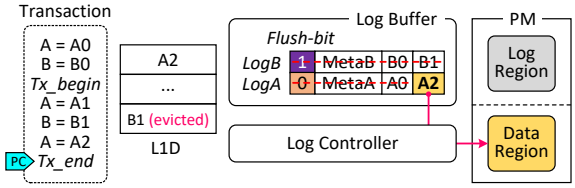

Fig. 8. Silo leverages the new data in logs to in-place update the data region.

Log Controller. We design a log controller in each memory controller to manipulate the logs in the log buffer. The log controller merges logs during transaction execution (§ III-C). After commit, the new data in log entries are flushed to the PM data region for in-place updates in common failure-free cases (§ III-D). These new data are coalesced in an on-PM buffer before being written to the PM media (§ III-E). Moreover, if a rare case occurs, e.g., a crash happens or the log overflows from the log buffer, the log controller flushes log entries to the PM log region for correct recovery (§ III-F and § III-G).

Log Region. In the log region, Silo leverages a distributed log scheme [49], in which each thread maintains its own log area to avoid contentions on writing logs across threads.

C. On-Chip Log Reduction

As the on-chip area is expensive, Silo leverages two schemes based on program behaviors to reduce the number of logs in the log buffer to alleviate the space overhead.

Log Ignorance. In some cases, a single CPU write does not actually modify the word, e.g., data copy and value assignment [47]. Due to no data change, it is unnecessary to generate a log entry. The log generator ignores this write.

Log Merging. Due to temporal locality in programs, multiple CPU writes could modify the data at the same physical address. In this case, we only need to record the *oldest* and *newest* data, since they are sufficient to recover the PM data to a *none* or *all* state after a crash. To this end, Silo merges the log entries that record the same data's changes into one log entry. Fig. 7 presents the log merging process. When a new log entry (e.g., $\texttt{Log}_{A1+A2}$) arrives, the log controller searches for an existing log entry that matches the *addr* in the new log entry. Specifically, the 64-bit comparator beside each log entry compares the *addr* in existing log entry with the *addr* in the new log entry. If a matched log entry is found (e.g., $\texttt{Log}_{A0+A1}$), Silo leverages the new data in $\texttt{Log}_{A1+A2}$ (i.e., A2) to replace the new data in $\texttt{Log}_{A0+A1}$ (i.e., A1), and then discards the new log entry. If there is no match, Silo appends the new log entry to the log buffer. This matching process in very fast since all the 64-bit comparators compare the *addrs* in parallel, and locate the matched log entry in less than 1 ns [1]. Silo merges logs without crossing threads or transactions. The merging operation is processed in the background without affecting the transaction execution.

D. Exploiting Logs for In-Place Updates

Based on our "Log as Data" idea, Silo leverages a *log-update scheme* to exploit logs to in-place update the data region in common failure-free cases. During transaction execution, the undo+redo logs recording the old+new data are stored in the log buffer. After commit, Silo flushes the *new data* in logs to update the data region, and then deletes other information in logs on chip. The old data in logs are only used in rare cases to guarantee the atomicity, as presented in § III-F and § III-G. Hence, Silo does not need to write the log entries to the log region, thus reducing the write traffic.

During transaction execution, Silo does not block any cache-line evictions. If a cacheline has been evicted to PM, flushing the new data in logs after commit causes unnecessary writes. To avoid this, Silo sets the *flush-bit* of the log entry to 1. It means that the new data in this log is simply discarded after commit. Once the write pending queue receives an evicted cacheline (e.g., `cl`) in transaction execution, the log controller simultaneously checks if there are logs that record the updates in `cl`. To achieve this, the comparators in the log buffer compare the line addresses of the log data (i.e., shifting the *addr* field) with the address of `cl`. If equal, the flush-bits of the matched logs are set to 1. All comparators work in parallel for fast comparisons. In consequence, Silo allows the new data in logs or cachelines to update the data region to reduce writes.

After a transaction (e.g., `Tx1`) commits, even if the modified cachelines in `Tx1` are evicted to PM, they will not incur redundant writes. The reason is that existing bit-level write reduction schemes, such as data-comparison-write [62], only apply the changed bits to the PM media. Since the new data of `Tx1` have been written to the data region, and the evicted cachelines contain the same words as these new data, these words will not be redundantly overwritten in the PM media, because the bits in these words are not changed.

Fig. 8 depicts the log-update scheme. During transaction execution, `LogA` is merged once. Data `B` has been evicted from cache and the flush-bit of `LogB` is set to 1. At commit, Silo only needs to flush the new data in `LogA` (i.e., `A2`) to the data region according to its *addr* in the log metadata.

Silo strictly guarantees the durability after the transaction commits. When the final update in a transaction completes, the log entry is sent to the log buffer in parallel with the next instruction execution. When `Tx_end` is executed, the log generator notifies the log controller that the transaction starts to commit. Hence, the final log entry arrives earlier than the commit notification. Once being notified, the log controller acknowledges the log generator and simultaneously flushes the new data. After the log generator receives the ACK, `Tx_end` completes and the CPU continues to execute the following codes. Such on-chip control message passing between the log generator and log generator only consumes several cycles [18]. After executing `Tx_end`, all logs containing the new data are in a persistent state, thus strictly guaranteeing the durability.

Our log-update scheme eliminates the ordering constraints,

i.e., the transaction commit does not need to wait for persisting the logs to the log region or persisting the updated cachelines to the data region. The reasons are threefold. 1) The logs are generated in parallel with cacheline updates and sent to the log buffer by bypassing the CPU caches. Hence, the logs are ensured to be persisted earlier than cachelines. 2) Silo uses logs for in-place updates, and thus it is unnecessary to wait for persisting logs to the log region before commit. 3) The transaction commit does not need to wait for flushing the new data, since these new data are not lost in the log buffer (backed by battery), thus enabling the data region to be lazily updated in the background. In consequence, Silo avoids the ordering constraints caused by writing the logs and the updated cachelines to PM.

In a multi-threaded scenario, Silo does not incur the synchronization overheads that involve cache coherence and multiple memory controllers (MCs), as analyzed below.

Cache Coherence. In Silo, the path of writing logs (i.e., from the log generator to the log controller) completely bypasses the CPU cache hierarchy. Hence, our logging scheme does not incur the overhead of cache coherence.

Multiple MCs. The efficiency of our log-update scheme is not affected by the number of MCs. When using multiple MCs, each MC serves the whole memory [30] and contains a log controller. A thread executes an entire transaction. The log generator sends the logs from the same transaction to the same MC. Hence, the logs and in-place updates end up at the same MC. In this way, it is unnecessary to coordinate different MCs to execute the same transaction, and Silo still reduces the overheads of log writes and ordering constraints.

E. Coalescing Writes to PM

Silo writes the new data in logs to PM in word granularity. In 64-bit CPUs, a word is 8B, which matches the 64-bit width of the processor-memory bus. Hence, each new data is atomically written to PM without wasting the bus width.

In general, a PM DIMM contains an internal buffer [50], [52], [55], [58] to temporarily store the written data to the underlying physical media for high access efficiency. We refer to this buffer as *on-PM buffer*, in which all the data will survive a crash by using ADR [58]. In Silo, the new data in logs are first written to the on-PM buffer before reaching the media. The line size of the on-PM buffer is larger (e.g., 256B [58]) than the size of new data (8B). Hence, these writes are applied to the media via read-modify-write operations, which could incur write amplification to the media [50], [55].

Silo leverages a *write coalescing scheme* in the on-PM buffer to reduce the write amplification to the media. This can be interpreted as three cases, as shown in Fig. 9. After commit, 6 words (`W1`–`W6`) are written to PM in order.

Case 1: `W1`–`W3` have the same physical line address in PM, but the bytes in `W3` overlap with those in `W1` and `W2`. Such overlapping cannot be eliminated by log merging since the physical addresses of these new data are different. However, Silo coalesces these writes in the on-PM buffer, i.e., the low 4B and high 4B in `W3` respectively overwrite the high 4B in `W1` and the low 4B in `W2`. The correctness is guaranteed since the log entries are added to the log buffer in order and these new data are flushed in the same order. The latter updated data will correctly overwrite the former one to ensure the freshness.

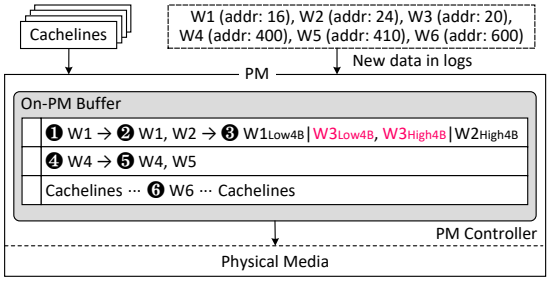

Fig. 9. The new data and cachelines are coalesced in the on-PM buffer.

Case 2: `W4` and `W5` have the same line address without overlapped bytes. Silo coalesces `W4` and `W5` by storing them together in the same line without writing the media twice.

Case 3: `W6` does not have the same line address with other new data. Silo stores `W6` with other cachelines. Silo allows the cachelines to share the on-PM buffer with the new data in logs. The 64B cachelines and 8B new data are coalesced in the same line to be written to the media together.

The two update paths in Fig. 9, i.e., cacheline eviction (CE) and in-place update (IPU) by using logs, do not cause any race risk, since the evicted cachelines and the new data in logs contain the *same* data values. To demonstrate the correctness, we summarize all three timing scenarios as follows.

1) *CE occurs earlier than IPU before commit:* The flush-bit in log is set to 1, and the log is discarded on chip. In this case, CE updates the PM data region.

2) *CE occurs in parallel with IPU during commit:* The new data in log is merged into the evicted cacheline in the on-PM buffer shown in Fig. 9. Hence, CE updates the PM data region.

3) *CE occurs later than IPU after commit:* Since the words in PM data region have already been updated by the new data in logs. These words would not be repeatedly updated by the evicted cacheline thanks to bit-level write reduction schemes [62], as mentioned in § III-D. In this case, IPU updates the words in the PM data region.

In summary, Silo orchestrates the two update paths, i.e., CE and IPU, to ensure that the PM data exist at the newest state and are never partially updated to guarantee the correctness.

F. Handling Log Overflow

If the log buffer cannot store all the logs in a large transaction, the log overflow occurs. Silo handles this by evicting the undo logs (FIFO) to the log region. 1) If the flush-bit of an evicted log is 1, the cacheline has been flushed. Silo flushes the undo logs to ensure atomicity, and the new data are discarded on chip. 2) If the flush-bit is 0, Silo sets it to 1 and flushes the undo logs to ensure atomicity, and simultaneously writes the new data to the data region to ensure durability. Hence, the log overflow does not break the atomic durability guarantee. The overflowed logs are deleted after commit if no crash occurs.

Silo flushes overflowed undo logs in a *batch* manner to improve the efficiency. We observe that if one log entry

overflows, the next log entry could also overflow. Hence, Silo flushes several undo logs at a time to make room for subsequent log entries. As the physical addresses of log entries are adjacent and each undo log entry is only 18B (including the log metadata and the old data), Silo flushes a batch of undo logs together, so that these logs can be stored in one line of the on-PM buffer to reduce the write amplification to the physical media. The number (N) of the batched log entries depends on the line size (S bytes) of the on-PM buffer, i.e., $N = \lfloor \frac{S}{18} \rfloor$. For example, if $S = 256$ [58], Silo writes 14 log entries in a batch upon a log overflow.

Log overflow does not heavily decrease the performance. The reasons are threefold. First, flushing the overflowed logs and adding subsequent logs to the log buffer are performed *in parallel*, which does not severely hamper the throughput. Second, there is no ordering constraint between flushing the undo logs and new data, since they are not lost in the log buffer and flushed to PM without any ordering requirement. Third, the write traffic does not significantly increase, since the overflowed logs are flushed in a batch manner to mitigate the write amplification to the PM media. The experimental results in § VI-F demonstrate the above analysis.

In general, log overflow is a rare case, since the write set of transactions is small (§ II-E), and Silo reduces the number of on-chip log entries (§ III-C). However, even if the log overflow occurs, Silo efficiently handles it without aborting transactions. Some prior schemes [25], [27] restrict the transaction size due to hardware limitations. Unlike them, Silo does not limit the transaction size to improve the generality and portability.

G. Selective Log Flushing for Crash Recovery

If a system crash or power failure occurs, Silo leverages a *selective log flushing scheme* to only flush the logs that are necessary for recovery according to the transaction states. 1) If a transaction fails to commit upon the crash, we need to ensure the atomicity to avoid partial updates in the data region. To achieve this, Silo flushes all the undo logs to the log region. The new data are discarded on chip. 2) If a transaction already commits upon a crash but the new data in logs have not been flushed, we need to guarantee the durability to ensure all the updates are not lost. To this end, Silo only flushes the redo logs whose *flush-bits are 0* and an ID tuple (tid, txid) to the log region. Silo leverages simple gates and multiplexers to implement the selective flushing logic in hardware. We use a small battery to supply the power to flush logs upon a crash.

After crashes, Silo recovers the data region by using logs. In the log region, the ID tuples record the committed transactions. For the remaining logs, Silo checks whether their (tid, txid) exist in the ID tuples. 1) If not, the logs are undo logs, which belong to the transactions that fail to commit. Silo reads the old data to revoke the partial updates in the data region. 2) If found, the transactions have committed and the logs are redo logs. Silo replays these redo logs to update the data region. In this case, even if some overflowed undo logs exist, Silo easily identifies and discards them, since the flush-bits in redo logs are 0. But in the overflowed undo logs, the flush-bits are 1.

TABLE I
THE HARDWARE OVERHEAD OF SILO.

Components	Types	Sizes
Log buffer	SRAM	20 entries, 680B per core
64-bit comparators	CMOS cells	20 comparators per log buffer
Battery	Lithium thin-film	$2.125\times10^{-4}mm^3$ per log buffer
Log head and tail	Flip-flops	16B per core

IV. PUTTING IT ALL TOGETHER

Fig. 10 illustrates an example on how our designs in § III work together to process transactions (Txs) and handle rare cases. Thread1 (T1) in core1 executes Tx1 and Tx3, and thread2 (T2) in core2 simultaneously executes Tx2. Data `A`–`H` are stored in different physical addresses. Their initial values are respectively `A0`–`H0` in PM.

- **Fig. 10a:** Data `A`, `B`, and `D` are updated. Their log entries are respectively stored in core1's and core2's log buffers.
- **Fig. 10b:** Tx1 commits. The new data in logs, i.e., `A1` and `B1`, in-place update the data region. Tx2 updates `E`.
- **Fig. 10c:** Tx2 updates `F`. The cacheline containing `D1` is evicted, and hence the flush-bit of `LogD` is set to 1.
- **Fig. 10d:** A new transaction (Tx3) updates `A` again. The log entry records `A1` as the old data and `A2` as the new data. Moreover, Tx2 updates `E` again. The log controller merges the logs to only record `E0` and `E2`.
- **Fig. 10e:** Tx3 updates `C` and `LogC` is created. In Tx2, the cacheline containing `F1` is evicted and `LogF`'s flush-bit is set to 1. Moreover, the undo log of `D` overflows.
- **Fig. 10f:** A crash or power failure occurs. Since Tx3 has committed, Silo flushes the redo logs with flush-bit of 0, and an ID tuple (T1, Tx3) to the log region. Moreover, Tx2 fails to commit and all the undo logs are flushed.
- **Fig. 10g:** During recovery, Silo identifies that Tx3 in T1 has committed, and hence all the log entries satisfying (tid = 1, txid = 3, flush-bit = 0) are the redo logs. Silo leverages them to replay transaction updates, i.e., `A1`→`A2` and `C0`→`C1`. Apart from these redo logs, other logs are undo logs that belong to the uncommitted transactions. Silo reads them to revoke the partially updated data, i.e., `D1`→`D0` and `F1`→`F0`.
- **Fig. 10h:** After recovery, the PM data region is in a consistent state. The updates in the committed transactions (Tx1 and Tx3) are persisted to guarantee durability. Moreover, the partial updates in the uncommitted transactions (Tx2) are discarded to guarantee atomicity.

Hardware Overhead. Table I summarizes the major hardware overhead of Silo in the processor. We adopt a 20-entry log buffer (i.e., 680B) for each core to store log entries from one transaction based on the results in § VI-D. A 64-bit comparator is associated with each log entry. Silo guarantees the persistence of log buffer by using batteries. Table IV shows that the required battery is very small. Moreover, Silo uses two 8B registers for each core to record the head and tail physical addresses of the thread-local log area in the PM log region.

Adding hardware components does not incur extra scheduling overheads due to two reasons. 1) Silo does not block or reorder the cacheline evictions, which are written to PM as

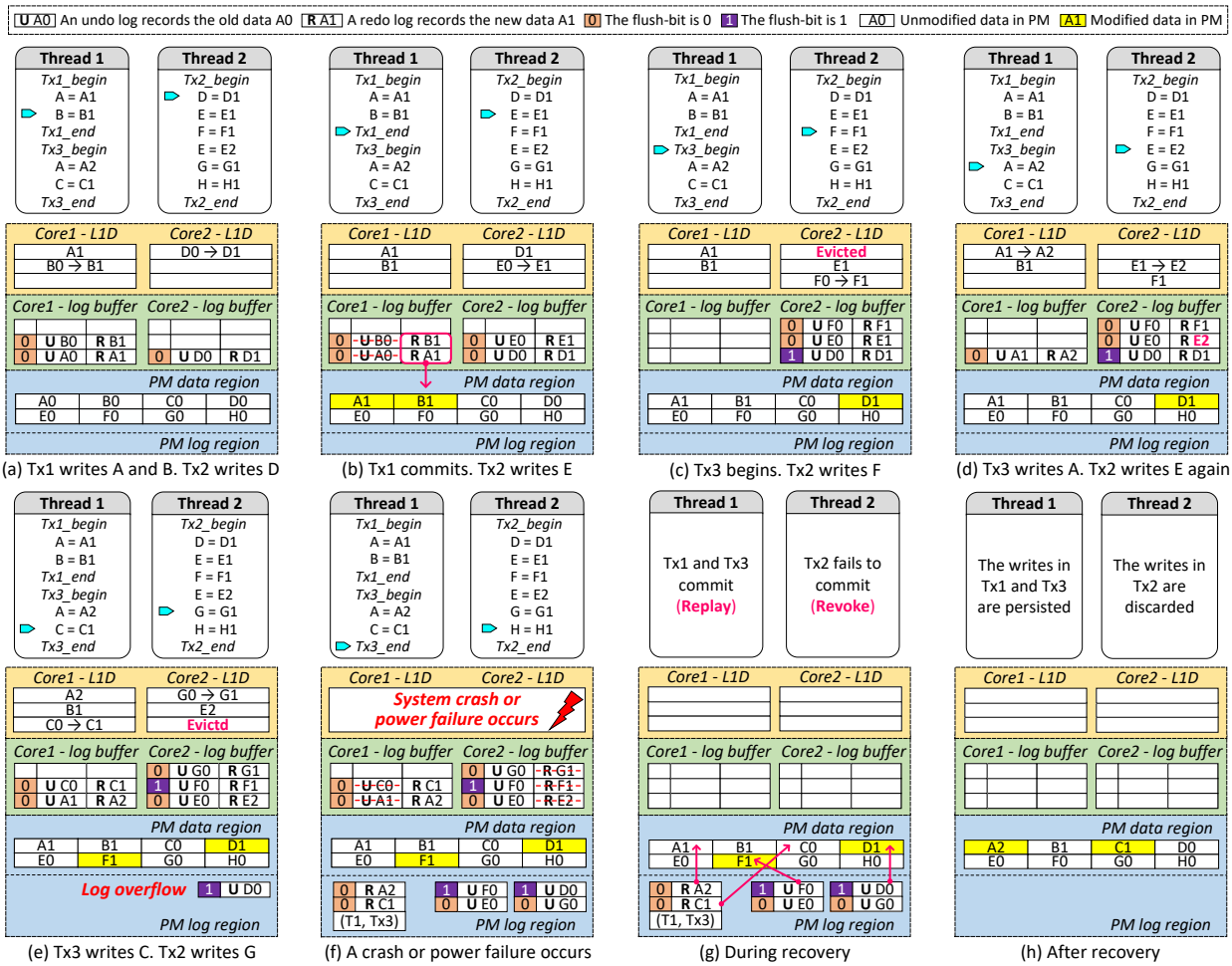

Fig. 10. An example of dealing with transaction execution, log overflow, and crash recovery in Silo.

unmodified systems. 2) The log entry records the *physical* data address. Hence, there is no address aliasing problem between processes. We do not need to save states on the context switch.

V. Discussions

eADR and BBB Using Battery-Backed Caches. eADR [22] and BBB [5] leverage batteries to make the CPU caches persistent. Hence, they support durability for single updates, i.e., these updates are forced to PM if a crash occurs. However, they do not guarantee atomicity. The data can be partially updated after a crash. Different from the goal of eADR and BBB, Silo supports both atomicity and durability for a group of updates. Table IV shows that Silo requires much smaller batteries than eADR/BBB to flush the on-chip logs on a crash.

Logless Atomic Durability. Instead of logging, LAD [18] buffers the updated cachelines in memory controller (MC) until committed to PM to guarantee atomic durability. LAD and Silo leverage MC as a persistent domain, but our logging design is significantly different from LAD in three aspects. 1) The transaction commit in LAD needs to wait for flushing the updated L1 cachelines to LLC and finally to MC, which stalls the CPU. Even if using an ideal proactive flushing scheme [26], [29], the transaction throughput improves by only 4% [18]. Unlike LAD, our log-update scheme enables the transaction to commit without waiting for flushing the updated cachelines. The results in Fig. 12 demonstrate our benefits. 2) LAD stores the entire cacheline in MC even if only one word is modified, which wastes the expensive space in MC and hence easily causes overflows. Unlike LAD, Silo stores log entries at the word granularity to improve the space efficiency. 3) If a cacheline overflow occurs, LAD needs to read the old data from PM to generate undo logs and flush them. These read operations incur high latency, causing LAD to fall back to a slow mode. Unlike LAD, Silo directly flushes the on-chip undo logs upon a log overflow without any PM read.

Hardware Logging Using Persistent Buffers. Some hardware logging studies adopt an on-chip persistent buffer (e.g., by using the ADR domain [44]) for different purposes. ATOM [28] uses the buffer to store the metadata to manage logs. MorLog [52] uses the buffer to reduce the persist latency for logs and data. ASAP [2] buffers the dependency lists and log headers. However, these studies still incur log writes to the log region in each transaction. Proteus [46] stores the undo logs on chip until commit. However, the transaction commit needs to wait for flushing the updated cachelines to cause ordering constraints. All these studies follow a traditional logging method to regard logs as backups, thus inevitably increasing the overheads even if using on-chip persistent buffers.

TABLE II
THE CONFIGURATIONS OF THE SIMULATED SYSTEM.

Processor	
Cores	8 cores, x86-64, 2 GHz
L1 I/D Cache	Private, 64B per line, 32KB, 8-way, 4 cycles
L2 Cache	Private, 64B per line, 256KB, 8-way, 12 cycles
L3 Cache	Shared, 64B per line, 8MB, 16-way, 28 cycles
Memory Controller	FRFCFS, 64-entry queue in ADR domain [44]
Log Buffer	680B per core, FIFO, 8 cycles, battery backed
Persistent Memory	
Capacity	16GB phase-change memory
Latency	Read / Write: 50 / 150 ns [10]

TABLE III
THE USED BENCHMARKS.

Micro-benchmarks [2], [27], [28], [38], [46], [52]	
Array	Randomly swap two elements in an array
Btree	Randomly insert elements in a B-tree
Hash	Randomly insert elements in a hash table
Queue	Randomly enqueue and dequeue elements in a queue
RBtree	Randomly insert elements in a red-black tree
Macro-benchamarks [25], [37], [52]	
TPCC	OLTP workload, all the five transaction types
YCSB	20%/80% of read/update for the key-value items

Unlike them, our Silo regards logs as data, and leverages the new data in on-chip logs to update the data region to make the common case fast. We use a persistent log buffer to ensure the logs can be flushed to PM for recovery in rare cases (e.g., crashes), which has a different purpose from the above studies.

VI. PERFORMANCE EVALUATION

A. Experiment Configurations

We leverage Gem5 simulator [9] with NVMain [43] to implement and evaluate Silo. The system configurations are shown in Table II. We run the benchmarks in Table III for performance comparisons. The micro-benchmarks are widely used in hardware logging studies [2], [27], [28], [38], [46], [52]. The size of data element is 64B in each micro-benchmark. The macro-benchmarks from Whisper [37] include two well-known real-world transactional workloads, i.e., `TPCC` and `YCSB`, which are configured like MorLog [52]: we run the `New-Order` transaction from `TPCC`, and set the read/update ratio in `YCSB` to 20%/80%. Considering different transaction types in real systems, we run all the five transaction types in `TPCC` to evaluate the capacity of log buffer in § VI-D. We use 8 cores (1 thread per core) to execute 10k transactions for each benchmark. The evaluated designs are shown below. ADR [44] is enabled for all the designs for fast persistency.

- **Base:** A hardware logging baseline that flushes an undo+redo log entry and the corresponding updated cacheline for each write.
- **FWB:** The logging design of FWB. The time interval of cache force write-back is set to 3,000,000 cycles [38].
- **MorLog:** The morphable logging scheme of MorLog. The delay-persistence commit protocol is disabled to ensure durability after the transaction commits [52].
- **LAD:** The logless atomic durability design of LAD [18]. LAD commits a transaction in two phases. The `Prepare` phase flushes the updated L1 cachelines to MC, and the `Commit` phase only sends messages. The proactive flushing scheme [26], [29] is enabled on LAD.

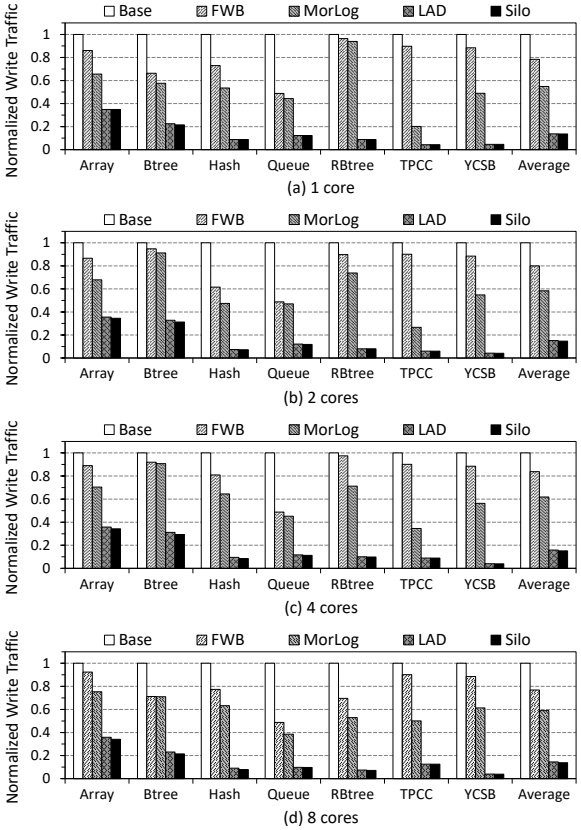

Fig. 11. The normalized write traffic to PM on different number of cores.

- **Our proposed Silo:** We reduce the number of logs in the log buffer (§ III-C), and leverage the log-update scheme to update the data region by using on-chip logs (§ III-D). The writes are coalesced in the on-PM buffer (§ III-E). Moreover, if a log overflow occurs, Silo flushes the overflowed logs in batch and in parallel (§ III-F).

Silo is not comparable with a recent study ASAP [2], since ASAP works on the customized *atomic region* instead of the ACID transaction. ASAP relaxes the durability requirement, i.e., the updated data and logs are not guaranteed to be persisted after the atomic region commits. In contrast, all the above evaluated designs strictly ensure the durability for transactions after commit. Moreover, FWB has demonstrated that it outperforms software (and hardware) undo or redo loggings. Hence, we do not repeatedly compare Silo with these schemes, since Silo outperforms FWB as shown in Fig. 11–12.

B. Write Traffic to PM

Fig. 11 shows the number of write requests to the PM physical media. The results are normalized to Base. Due to flushing the log and modified cacheline for each write, Base suffers from the highest write traffic. MorLog and FWB back up data to the log region, which incurs redundant writes. Their log metadata also increase the number of writes. MorLog reduces the writes by 30% over FWB due to mitigating the intermediate redo logs. LAD exhibits low write traffic due to only writing cachelines. By using logs to update the data region, Silo avoids writing logs to the log region. Moreover, by coalescing the writes of new data in the on-PM buffer, Silo

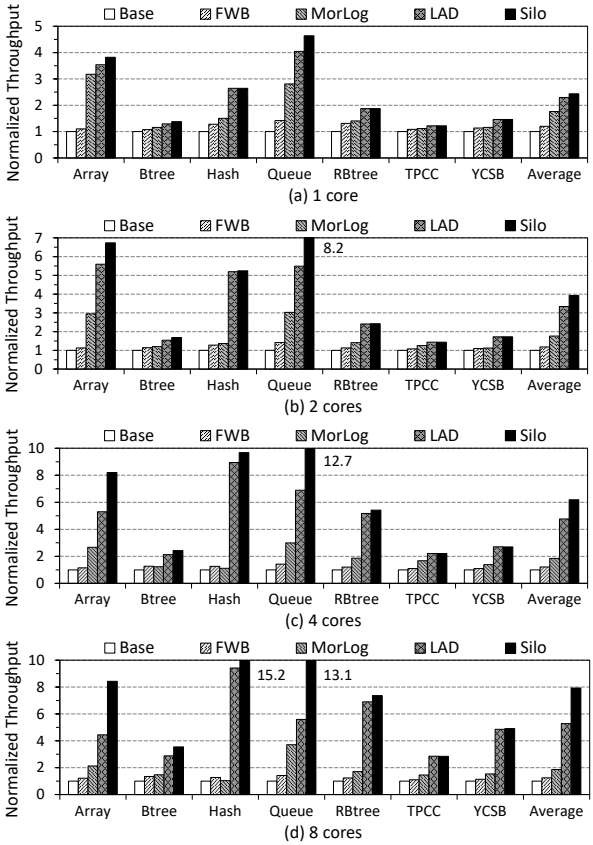

Fig. 12. The normalized transaction throughput on different numbers of cores.

alleviates the write amplification to the PM media, and hence exhibits approximate write traffic with LAD. Silo reduces the writes by 76.5%/82% over MorLog/FWB on 8 cores, thus significantly improving the lifetime of PM.

C. Transaction Throughput

Fig. 12 presents the transaction throughput of all benchmarks running on 1–8 cores. The results are normalized to Base, which shows the lowest throughput due to the heavy overheads on logging and cacheline flushes. On 8 cores, MorLog outperforms FWB by 1.5× via reducing the intermediate redo data. Due to not producing logs, LAD shows higher throughput than FWB and MorLog.

When using more CPU cores, Silo achieves higher throughput improvements, since our log-update scheme removes the ordering constraints to improve the scalability. On 8 cores, Silo respectively improves the throughput by 1.5×/4.3×/6.4× compared with LAD/MorLog/FWB. Although the write traffic of LAD is close to that of Silo, LAD shows lower throughput than Silo, since LAD suffers from a long *write path* to commit the data updates (i.e., L1 → LLC → MC). In other words, LAD consumes more time to wait for flushing the modified L1 cachelines to MC before commit, while Silo does not need to wait for flushing any cacheline, thus accelerating the commit. Moreover, Silo significantly outperforms LAD on `Array` and `Queue`, since these workloads exhibit low spatial locality, causing many dirty cachelines per transaction. As a result, LAD needs to wait for flushing substantial dirty cachelines to MC before commit, thus decreasing the performance.

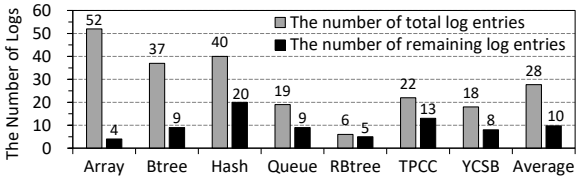

Fig. 13. The number of total and remaining on-chip logs per transaction.

TABLE IV
THE BATTERY REQUIREMENTS OF DIFFERENT SYSTEMS (8 CORES).

	eADR [22]	**BBB** [5]	**Our Silo**
Flush Size (KB)	10,496	16	5.3125
Flush Energy (μJ)	54,377	194	62
Cap (mm^3; mm^2)	151; 28.4	0.54; 0.66	0.17; 0.31
Li (mm^3; mm^2)	1.51; 1.32	0.0054; 0.031	0.0017; 0.014

D. The Capacity of Log Buffer

To evaluate the capacity of the log buffer, we check the number of on-chip logs per transaction. Fig. 13 shows the number of the total and remaining log entries per transaction of each core. Our log reduction schemes in § III-C mitigate 64.3% of the logs on average. `Array` produces many logs, since each data element is 64B, which involves multiple words. However, many words are not actually modified and 90.4% of logs are ignored. As the maximum number of remaining log entries is 20 in `Hash`, we reserve 20 entries in each core's log buffer. In 64-bit CPUs, each undo+redo log entry is 26B, and has an 8B physical address. Hence, we configure the capacity of the log buffer for each core to be 680B (i.e., $20 \times (26+8)$). In our 8-core configuration, the total buffer size is 5,440B.

E. Energy and Battery Requirements for Log Buffer

We leverage the battery-backed SRAM to implement the log buffer. To evaluate the energy for flushing the on-chip logs after a crash, we use the energy consumption model from [5], [41], i.e., moving one byte from the log buffer to PM consumes 11.228 nJ. Hence, we require 62 μJ to flush a 5,440B log buffer. To supply the required energy, we evaluate two types of batteries, i.e., supercapacitors (Cap) [63] and lithium thin-film batteries (Li) [42]. The energy density of Cap/Li is 10^{-4}/10^{-2} Wh cm^{-3} [54]. We hence obtain the volumes (mm^3) and areas (mm^2 in cubic shapes) of Cap and Li, as shown in Table IV.

For comparisons, Table IV also presents the required energy and battery for BBB [5] and eADR [22], although their design goals are different from Silo, as discussed in § V. BBB flushes the 16KB buffers of 8 cores (each core has 32 64B entries). eADR flushes the dirty blocks (45%) in the 10,496KB CPU caches in Table II. The results show that eADR/BBB consume 888.2×/3.2× (91.6×/2.1×) larger volume (area) of Cap than Silo. Hence, the battery overhead of our log buffer is very low.

F. Processing Large Transactions

In rare cases, the logs could overflow from the log buffer when processing large transactions. To study how Silo performs in large transactions, we set the write set of a transaction to be 1–16× larger than the size of log buffer. For each benchmark, the transaction throughput and PM write traffic are normalized to the 1× configuration. Fig. 14a shows that the throughput decreases by only 7.4% on average when processing 16× larger transactions, since Silo enables the

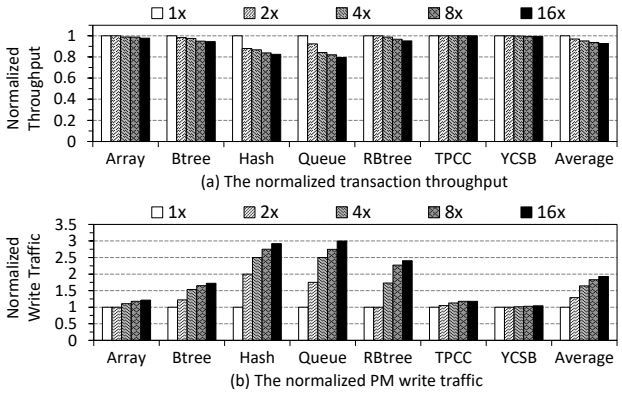

Fig. 14. The transaction performance on different sizes of the write set.

overflowed logs to be flushed in parallel with generating new logs. Fig. 14b shows that the write traffic only increases by up to 1.9× on average since Silo flushes the overflowed undo logs in a batch manner to mitigate the write amplification to PM media. The performance on `Btree`, `Hash`, `Queue`, and `RBtree` decrease when running large transactions due to writing extra overflowed logs. Note that `Array` shows stable performance since most of the logs are ignored as analyzed in § VI-D. Hence, the logs do not frequently overflow. Moreover, the results on `TPCC` and `YCSB` keep stable due to their good locality, which enables substantial logs to be merged on chip. In summary, the log overflow does not always occur in large transactions. Even if it occurs, Silo does not abort transactions or incur severe performance degradations.

G. Performance Sensitivity to the Latency of Log Buffer

We study how the access latency of the log buffer affects the performance. We change the latency from 8 to 128 cycles to cover various buffer types (e.g., SRAM). The throughputs of micro-/macro-benchmarks are normalized to `Array`/`TPCC` using an 8-cycle buffer. Fig. 15 shows that the throughput generally keeps stable when increasing the latency. In Silo, the CPU `store` does not need to wait for writing logs to the buffer during transaction execution, and the new data in logs are read from the buffer to update the data region in the background after commit. Thus, reading or writing the log buffer is not on the critical path. Using a 128-cycle buffer only decreases the throughput by 3.3% over an 8-cycle one on average. Moreover, the write traffic is not affected when changing the latency. In summary, the latency of log buffer has negligible effect on the efficiency of Silo.

VII. Related Work

WAL for Atomic Durability. Software loggings [12], [14], [48], [56] rely on CPU instructions to enforce the durability order between logs and data. DudeTM [34] and SoftWrAP [17] use a DRAM cache to remove the persist operations from the critical path, but need to track the data versions. Unlike these studies, Silo adopts the hardware logging approach.

Hardware logging efficiently overlaps the log operations and transaction execution. Prior hardware undo loggings [28], [46] need to persist all the updated data before commit. ASAP [2] asynchronously persists the undo logs and the updated data

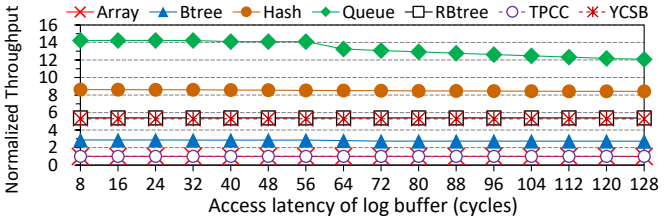

Fig. 15. The normalized transaction throughput on different buffer latencies.

after commit, but need to track the data and control dependencies. Existing redo schemes [16], [25], [27], [51] enforce the ordering between redo logs and data. DHTM [27] writes redo logs to provide durability for hardware transactional memory, but the transaction size is limited by LLC. CCHL [51] compresses and consolidates logs to reduce writes. Legacy undo+redo designs [38], [52] exploit the benefits of undo and redo loggings, but still write extra logs. Unlike them, Silo uses the on-chip logs to directly in-place update the data region in common failure-free cases, thus reducing the overheads.

Multi-Versioning Schemes for Atomic Durability. Atomic durability can be ensured by multi-versioning [10], [18], [35], [61]. Kiln [61] uses a non-volatile last level cache (NVLLC) to store the updated data. LAD buffers the updated cachelines in memory controller until committed to PM. Kamino-Tx [35] maintains the main and backup versions of data regions in PM. HOOP [10] designs an indirection layer that redirects the addresses for out-of-place updates. Unlike them, Silo adopts hardware logging to ensure atomic durability, while enabling in-place updates without the needs of NVLLC, data region backups, and physical address redirections.

Crash Consistency for Single Operations. Some studies guarantee the crash consistency for single operations on PM. They can be divided into two categories. First, the software-based data structures, such as NVTree [59], Fast&Fair [20], Level Hashing [64], and MOD [19], leverage customized schemes to ensure the consistency for single updates. Second, the hardware-based schemes, such as eADR [22] and BBB [5], adopt battery-backed caches to persist CPU writes. Orthogonal to these studies, our Silo focuses on the atomic durability for a group of updates based on the ACID transaction.

VIII. Conclusion

In order to ensure atomic durability for persistent memory (PM), this paper proposes Silo, a speculative hardware logging approach that leverages the new data in the on-chip logs to in-place update the PM data region in common failure-free cases. Hence, it is unnecessary to write logs to the PM log region to back up data, thus improving the performance and reducing the overheads. Only in rare cases, e.g., system crashes, Silo selectively flushes necessary on-chip logs to PM for data recovery without any loss of correctness. Experimental results demonstrate that Silo significantly outperforms state-of-the-art studies in terms of transaction throughput and write traffic.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022, and Key Laboratory of Information Storage System, Ministry of Education of China.

REFERENCES

[1] S. Abdel-Hafeez, A. Gordon-Ross, and B. Parhami, "Scalable digital cmos comparator using a parallel prefix tree," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 11, pp. 1989–1998, 2013.

[2] A. H. M. O. Abulila, I. E. Hajj, M. Jung, and N. S. Kim, "ASAP: architecture support for asynchronous persistence," in *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*. ACM, 2022, pp. 306–319.

[3] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010.

[4] M. Alomari, M. Cahill, A. Fekete, and U. Rohm, "The cost of serializability on platforms that use snapshot isolation," in *2008 IEEE 24th International Conference on Data Engineering*, 2008.

[5] M. A. Alshboul, P. Ramrakhyani, W. Wang, J. Tuck, and Y. Solihin, "BBB: simplifying persistent programming using battery-backed buffers," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 2021, pp. 111–124.

[6] M. A. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*. IEEE Computer Society, 2018, pp. 439–451.

[7] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer torque magnetic random access memory (STT-MRAM)," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, pp. 13:1–13:35, 2013.

[8] L. Benson, L. Papke, and T. Rabl, "Perma-bench: Benchmarking persistent memory access," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2463–2476, 2022.

[9] 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, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[10] M. Cai, C. C. Coats, and J. Huang, "HOOP: efficient hardware-assisted out-of-place update for non-volatile memory," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 584–596.

[11] D. R. Chakrabarti, H. Boehm, and K. Bhandari, "Atlas: leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. ACM, 2014, pp. 433–452.

[12] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "REWIND: recovery write-ahead system for in-memory non-volatile data-structures," *Proc. VLDB Endow.*, vol. 8, no. 5, pp. 497–508, 2015.

[13] Z. Chen, Y. Hua, Y. Zhang, and L. Ding, "Efficiently detecting concurrency bugs in persistent memory programs," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 2022, pp. 873–887.

[14] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. ACM, 2011, pp. 105–118.

[15] Z. Dang, S. He, P. Hong, Z. Li, X. Zhang, X. Sun, and G. Chen, "Nvalloc: rethinking heap metadata management in persistent memory allocators," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 2022, pp. 115–127.

[16] K. Doshi, E. Giles, and P. J. Varman, "Atomic persistence for SCM with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE Computer Society, 2016, pp. 77–89.

[17] E. Giles, K. Doshi, and P. J. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*. IEEE Computer Society, 2015, pp. 1–14.

[18] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 466–478.

[19] S. Haria, M. D. Hill, and M. M. Swift, "MOD: minimally ordered durable datastructures for persistent memory," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 2020, pp. 775–788.

[20] D. Hwang, W. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*. USENIX Association, 2018, pp. 187–200.

[21] IBM, "Telecom application transaction processing (tatp) benchmark," http://tatpbenchmark.sourceforge.net/, 2011.

[22] Intel, "eADR: New Opportunities for Persistent Memory Applications," https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[23] Intel, "Intel® Optane™ Persistent Memory," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2022.

[24] Intel, "The libpmemobj library in Intel Persistent Memory Development Kit," https://pmem.io/pmdk/libpmemobj/, 2022.

[25] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, pp. 520–532.

[26] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. ACM, 2015, pp. 660–671.

[27] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "DHTM: durable hardware transactional memory," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*. IEEE Computer Society, 2018, pp. 452–465.

[28] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. IEEE Computer Society, 2017, pp. 361–372.

[29] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger, "Viyojit: Decoupling battery and DRAM capacities for battery-backed DRAM," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 2017, pp. 613–626.

[30] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *2010 IEEE International Symposium on High Performance Computer Architecture, HPCA 2010, Bangalore, India, January 9-14, 2010*. IEEE Computer Society, 2010, pp. 1–12.

[31] H. Kimura, "FOEDUS: OLTP engine for a thousand cores and NVRAM," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 691–706.

[32] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: converting concurrent DRAM indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 2019, pp. 462–477.

[33] Y. Lee, S. Kim, S. Hong, and J. Lee, "Skinflint DRAM system: Minimizing DRAM chip writes for low power," in *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*. IEEE Computer Society, 2013, pp. 25–34.

[34] 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 Twenty-Second International Conference on Architectural Support for Programming Languages and Operating*

Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017. ACM, 2017, pp. 329–343.

[35] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. ACM, 2017, pp. 499–512.

[36] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, 1992.

[37] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with WHISPER," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. ACM, 2017, pp. 135–148.

[38] M. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+redo logging for persistent memory systems," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 2018, pp. 336–349.

[39] Oracle, "NVM-Direct Library," https://github.com/oracle/nvm-direct, 2015.

[40] Oracle, "What is OLTP?" https://www.oracle.com/database/what-is-oltp/, 2022.

[41] D. Pandiyan and C. Wu, "Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms," in *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*. IEEE Computer Society, 2014, pp. 171–180.

[42] D. Pech, M. Brunet, H. Durou, P. Huang, V. Mochalin, Y. Gogotsi, P.-L. Taberna, and P. Simon, "Ultrahigh-power micrometre-sized supercapacitors based on onion-like carbon," *Nature nanotechnology*, vol. 5, no. 9, pp. 651–654, 2010.

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

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

[45] A. Rudoff, "Persistent memory programming without all that cache flushing," *SDC*, 2020.

[46] 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, Cambridge, MA, USA, October 14-18, 2017*. ACM, 2017, pp. 178–190.

[47] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level cache deduplication," in *2014 International Conference on Supercomputing, ICS'14, Muenchen, Germany*. ACM, 2014, pp. 53–62.

[48] 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, Newport Beach, CA, USA, March 5-11, 2011*. ACM, 2011, pp. 91–104.

[49] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endow.*, vol. 7, no. 10, pp. 865–876, 2014.

[50] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 2020, pp. 496–508.

[51] X. Wei, D. Feng, W. Tong, J. Liu, C. Wang, and L. Ye, "CCHL: compression-consolidation hardware logging for efficient failure-atomic persistent memory updates," in *ICPP 2020: 49th International Conference on Parallel Processing, Edmonton, AB, Canada, August 17-20, 2020*. ACM, 2020, pp. 12:1–12:11.

[52] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye, "Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 610–623.

[53] H. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[54] Z.-S. Wu, K. Parvez, X. Feng, and K. Müllen, "Graphene-based in-plane micro-supercapacitors with high power and energy densities," *Nature communications*, vol. 4, no. 1, pp. 1–8, 2013.

[55] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, "Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering," in *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 2022, pp. 488–505.

[56] Y. Xu, J. Izraelevitz, and S. Swanson, "Clobber-nvm: Log less, re-execute more," in *ASPLOS '21: Architectural Support for Programming Languages and Operating Systems, 2021*. ACM, 2021.

[57] S. Yadalam, N. Shah, X. Yu, and M. Swift, "ASAP: A speculative approach to persistence," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 2022, pp. 892–907.

[58] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. USENIX Association, 2020, pp. 169–182.

[59] 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 of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*. USENIX Association, 2015, pp. 167–181.

[60] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1567–1581.

[61] 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 *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*. ACM, 2013, pp. 421–432.

[62] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*. ACM, 2009, pp. 14–23.

[63] Y. Zhu, S. Murali, M. D. Stoller, K. J. Ganesh, W. Cai, P. J. Ferreira, A. Pirkle, R. M. Wallace, K. A. Cychosz, M. Thommes, D. Su, E. A. Stach, and R. S. Ruoff, "Carbon-based supercapacitors produced by activation of graphene," *Science*, vol. 332, no. 6037, pp. 1537–1541, 2011.

[64] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 2018, pp. 461–476.