SpecPMT: Speculative Logging for Resolving Crash Consistency Overhead of Persistent Memory

Chencheng Ye*
Huazhong University of Science and Technology
Wuhan, Hubei, China
yecc@hust.edu.cn

Yuanchao Xu
North Carolina State University
Raleigh, North Carolina, USA
yxu47@ncsu.edu

Xipeng Shen
North Carolina State University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

Yan Sha*
Huazhong University of Science and Technology
Wuhan, Hubei, China
soyan0408@gmail.com

Xiaofei Liao*
Huazhong University of Science and Technology
Wuhan, Hubei, China
xfliao@hust.edu.cn

Hai Jin*
Huazhong University of Science and Technology
Wuhan, Hubei, China
hjin@hust.edu.cn

Yan Solihin
University of Central Florida
Orlando, Florida, USA
Yan.Solihin@ucf.edu

ABSTRACT

Crash consistency overhead is a long-standing barrier to the adoption of byte-addressable persistent memory in practice. Despite continuous progress, persistent transactions for crash consistency still incur a 5.6× slowdown, making persistent memory prohibitively costly in practical settings. This paper introduces *speculative logging*, a new method that forgoes most memory fences and reduces data persistence overhead by logging data values early. This technique enables a novel persistent transaction model, *speculatively persistent memory transactions* (SpecPMT). Our evaluation shows that SpecPMT reduces the execution time overheads of persistent transactions substantially to just 10%.

CCS CONCEPTS

• **Software and its engineering → Software reliability**.

KEYWORDS

persistent memory, transaction, logging, microarchitecture

ACM Reference Format:
Chencheng Ye, Yuanchao Xu, Xipeng Shen, Yan Sha, Xiaofei Liao, Hai Jin, and Yan Solihin. 2023. SpecPMT: Speculative Logging for Resolving Crash Consistency Overhead of Persistent Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3575693.3575696

*Chencheng Ye, Yan Sha, Xiaofei Liao, and Hai Jin are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575696

1 INTRODUCTION

Byte-addressable persistent memory (e.g., PCM [19]) features high density, byte-addressability, and data persistency [70, 78], allowing software to access durable data in main memory. Experiments in cloud services [17, 18, 20, 40, 83] and high-performance computing [9, 23, 68] have demonstrated the potential of persistent memory in building crash-resilient software, improving software performance, and increasing development efficiency.

Despite the promising potential, *crash consistency overheads* impose a barrier to the widespread adoption of persistent memory. Crash consistency is essential for a program to maintain a consistent state of persistent data in memory across crashes. It is a fundamental requirement for the recoverability and reusability of persistent data and the resumption of execution. Programmers often use *persistent memory transactions* to achieve crash consistency — transactions provide simple yet powerful semantics of crash-atomic updates, i.e., they ensure either all or no transactional updates on persistent memory locations are observable after a crash.

Existing persistent memory transactions, however, incur large overheads, because of the need to log data updates to provide transactional semantics. For example, the most commonly used implementation, Intel PMDK, was reported to incur 6× slowdowns to program executions [30, 71]. This *crash consistency overhead* must be substantially lowered before persistent memory becomes attractive for wide adoption.

This important problem has been the focus of recent studies [5, 13, 14, 63, 69, 71, 76]. Some of the proposed solutions are specially designed for a certain algorithm or a data structure [9, 52], and are

not applicable to general applications. More general solutions include pure software methods [14, 71] and hardware support [13, 63] to mitigate the logging overhead. Although these studies have made important contributions and reduced the logging overheads by as much as 2.6× [25, 71], crash consistency overheads are still too large for practical use. As shown in Figure 1, even after applying state-of-the-art software (SPHT [14]) or hardware (EDE [63]) solutions, the programs in STAMP [53] still suffer from an average of 50–161% execution time overheads compared to versions without persistent memory transactions. Crash consistency overheads remain an unsolved problem that presents a barrier to practical adoption.

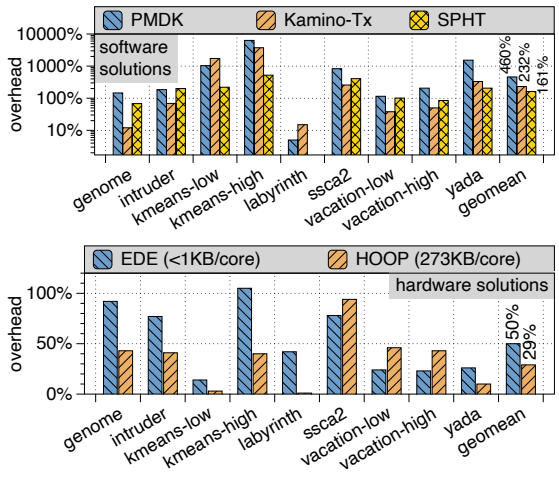

Figure 1: After applying the state-of-the-art schemes, programs still suffer from significant slowdowns over versions without persistent memory transactions. Software solutions were evaluated for STAMP benchmark suite [53] on a real machine (above) and on a simulated hardware (below). Details of the experimental setup are in Section 7.

In this work, we present *speculative logging*, a novel approach to reduce crash consistency overheads. The design was motivated by the fact that the main sources of crash consistency overheads in persistent transactions are the use of memory fences and the persisting of data. The design builds upon the *key insight* that data can be speculatively logged early, and doing so removes the sources of performance overheads.

Figure 2 illustrates the basic idea of speculative logging. In a typical persistent transaction (Figure 2 left), a datum is (undo) logged before being updated in the transaction. A flush and fence ensure that the log write persists before the data write. In contrast, speculative logging (Figure 2 right) moves up the logging to a point as early as the last transaction where the datum was updated. By doing that, several benefits are achieved. First, the log write leverages the commit of the first transaction to persist the log, forgoing the need for memory fences for persisting the log. Second, it defers the flush to the transaction commit, making the transaction execute faster. Third, as the log persists the most recent value of the datum once a transaction commits, the data write in the same transaction

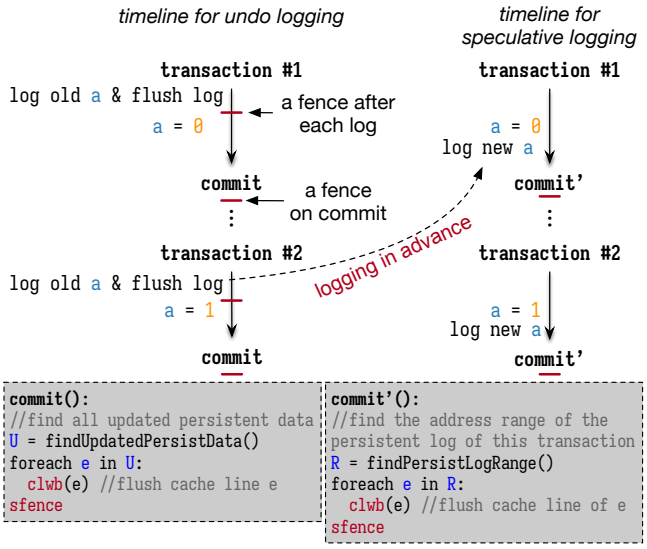

Figure 2: Timeline for two consecutive persistent transactions. The undo logging transaction (left) records the old value of a datum, using multiple fences in a transaction. Speculative logging transaction (right) records the new value of the memory location without fences. It persists all log records but data with only one fence each transaction before the transaction commits.

becomes optional. If the data write fails to persist, the post-crash recovery can rely on log records to rebuild the updated data. Thus, the data write no longer requires a flush [54]. Because persisting log records involve sequential writes, they have better spatial locality and are faster than persisting data writes (which may be more random [78]).

Turning this basic idea into a full solution is not trivial. Several challenges arise for both efficiency and recoverability: How should the log record be formatted and maintained to enable correct recovery when a crash happens either inside or outside a transaction? Because the speculative log records must outlive the transaction commit, how long should the records be kept in the presence of multiple transactions and repeated data updates? How can useless log records be identified and reclaimed in a timely and non-blocking manner?

This paper addresses these questions with two schemes. The first solution is a software-only solution and is compatible with existing hardware. It uses novel speculative log management featuring a compact log format and background memory reclamation, and a crash recovery protocol based on the compact log format. The proof-of-concept implementation of the solution achieves about 2.7× speedup over the state-of-the-art in-place solution Kamino-Tx [51]. However, two key drawbacks remain: (1) memory space overheads, which is 3× persistent memory space, and (2) the reliance on dedicated background memory reclamation threads.

To reduce memory space overheads while preserving performance, we propose a second solution with a novel architecture for a hybrid logging model. The model allows softwares to use

speculative logging for *hot* (i.e., frequently updated) data and undo logging for cold data. By controlling the threshold of data hotness, the user can set an arbitrary bound on the size of the speculative log area. The hybrid logging model also enables a software-hardware co-designed *log reclamation scheme*. The log reclamation runs in the foreground, and therefore does not require dedicated reclamation threads. This allows softwares to reclaim speculative log records with a few instructions without blocking other running threads. Specifically, the speculative log records are sorted in chronological order and divided into epochs. To track the epoch in which a log record is created, a few status bits are added to private TLB entries. By simply clearing the bits associated with a given epoch with new instructions, the hardware can easily reclaim all the log records created in the epoch.

Together, these solutions provide an alternative approach to the current undo logging–based persistent transactions, namely *speculatively persistent memory transactions (SpecPMT)*. Comparisons with state-of-the-art in-place update methods (Kamino-Tx [51] and EDE [63]) show that SpecPMT can reduce the execution time overheads of the prior methods from 232% and 50% to 10% and 7%, respectively.

SpecPMT achieves all the desirable properties of a persistent memory transaction while keeping transaction overheads low: (1) it uses in-place data updates; (2) it eliminates fences between logging and data updates; (3) it does not block transaction commit with data persistence; (4) it supports a software-only or a lightweight hardware implementation; and (5) it is data structure agnostic rather than data structure specific.

To summarize, the contributions of this work are:

- We propose *speculative logging* for substantially reducing crash consistency overheads of persistent transactions based on the removal of fences.
- We propose a variant of SpecPMT for software-only speculative logging, which uses a novel log organization, recovery protocol, and log reclamation.
- We provide a mechanism for hardware-supported speculative logging with SpecPMT, which uses a novel hybrid logging scheme and an epoch-based log reclamation that bound the memory space cost without background reclamation threads.
- We empirically evaluate the effectiveness of SpecPMT.

2 BACKGROUND

This section covers the background on crash consistency of persistent memory.

2.1 Persistent Memory

Non-volatile memory (NVM) or storage class memory represents an array of new memories that provide byte-addressability, persistency, random access, and DRAM-like access latencies. NVM can be implemented by simply adding a battery to DRAM [44] or exploiting new memory technologies such as PCM [19] or FeRAM [61]. Persistent memory [59] refers to the use of NVM for storing data structures so that they can continue to be accessed beyond a process lifetime, even across crashes or system boots.

Intel Optane DC persistent memory [59] is currently the most widely available substrate for persistent memory. It is manufactured as a *dual in-line memory module* (DIMM) and should be attached to the memory bus of Intel x86 platforms.

Recently, researchers have made substantial progress in building new persistent memory devices [4], utilizing *compute express link* (CXL) as an alternative point for attaching new memory-semantic SSDs [43] or upcoming storage class memory devices. Those devices are expected to interface with the programmers similarly to how Intel Optane DC persistent memory is used in the local node. Without loss of generality, in this paper, we assume the Intel Optane DC persistent memory hardware model.

2.2 Memory Data Persistency

Modern processors rely on a volatile SRAM cache hierarchy to accelerate data access. A store to a location may stay in a volatile cache for a long time, only to be persisted on cache eviction. Thus, managing persistency in software requires hardware support that allows the software to specify when data should be persisted.

Since the Skylake architecture, Intel has provided a `CLWB` instruction to flush a dirty cache line into memory explicitly (in addition to `CLFLUSH` and `CLFLUSHOPT`). Such an instruction is often followed by `SFENCE` to complete a persist barrier (i.e., to prohibit younger stores/flushes from persisting until all older ones have persisted). A store/flush is considered persisted when it reaches the persistence domain, which takes thousands of CPU cycles [67, 70, 78]. SpecPMT elides such cost when it removes the `SFENCE`.

In systems that adhere to *asynchronous DRAM refresh* (ADR) requirements, the persistence domain includes the main memory and memory controller *write pending queue* (WPQ). With *extended asynchronous DRAM refresh* (eADR), the persistence domain extends to CPU caches, removing the need for cache line flushes. But eADR has not received wide adoption as discussed in Section 5.3.

Another issue for memory data persistence is the need for a system abstraction for persistent data. Because persistent data must survive across system power cycles, applications must be able to find the durable data created before the current power cycle. The SNIA NVM programming model [59] organizes durable data as persistent memory-mapped files and manages the files with a persistent memory file system. To disable the application-transparent DRAM page cache that buffers reads and writes to the files, a persistent memory-aware file system can grant applications direct file accesses, e.g., via *direct file access* (DAX) [2]. Alternatively, with appropriate OS support, persistent data can be kept in a file-less persistent memory object abstraction [27, 79].

2.3 Crash Consistency

An application may use persistent memory transactions [50, 57, 66] to achieve crash consistency. A persistent memory transaction guarantees that writes in the transaction are atomically (either entirely or not at all) observable at recovery. Transactions use logging mechanisms such as write-ahead logging [13, 42, 62] or shadow memory [56] to realize such atomic durability. A write-ahead logging mechanism creates and persists a log record before the datum is persisted. The recovery uses the log to revoke the effect of the datum update. To enforce the ordering between log record

and data persistence, programmers use a persist barrier (e.g., flush the log record with CLWB followed by a memory fence instruction, SFENCE).

Write-ahead logging schemes may rely on undo or redo logging. With undo logging [42, 57, 62, 63], the transaction records the old value of data before it is updated in place (i.e., at the location). If the application crashes before the transaction commit, the recovery restores data to the logged value. In contrast, with redo logging [35, 46, 51], the transaction buffers all write intents in log records. Any accesses to those memory locations are redirected to the logged records for the updated value before the transaction commits. At commit, the transaction persists all the logged write intents, then applies the write intents to the corresponding memory locations. The recovery discards all log records if the application crashes before all write intents reach the persistence domain. Otherwise, the recovery reapplies the persisted write intents to the memory locations.

A transaction may also provide concurrency control, which is orthogonal to the crash consistency mechanism. For simplicity, this paper focuses on SpecPMT durability control but briefly discusses concurrency control in Section 4.3.

3 SPECULATIVE LOGGING

SpecPMT is an optimization on undo logging that improves its performance through the amortization of persistent memory ordering instructions.

Classical undo logging ensures correct recovery by ensuring that for every location modified by a transaction: (1) a log entry exists recording its old value, and (2) this log entry becomes persistent before the location is modified. Following these steps means that if a transaction is interrupted and power is lost, any locations modified by the failed transaction can be rolled back to their original values.

Our observation is that while undo logging performs both these steps during the transaction, they need not be, and for persistent memory, there are performance advantages to migrating them earlier.

First, we create the log entry far earlier than the modifying transactions, e.g., before entering the transaction or in a prior transaction. Second, it allows the software to discard a log record once the associated datum reaches the persistent domain and rebuild the record at any point before the datum gets updated again.

SpecPMT adopts a novel strategy based on the above observation by maintaining a speculative log record for every single datum. SpecPMT preserves classical persistent memory transactional APIs [59, 66] for compatibility, as shown in Figure 3, with regular logging replaced by speculative logging (*splog*). The recovery API is omitted from the example codelet as it is only needed for post-crash recovery.

Inside a transaction, the programmer or the compiler inserts *splog* after each durable data update to create a log record for the virtual address of the datum and the new value of the datum. No cache line flush or fence is needed at this point. Instead, the transaction persists all log records by flushing them and using a single store fence before commit.

A crash causes two kinds of data corruption: (1) data updated by uncommitted transactions reaching persistent memory, and (2) data

tx_begin()
annotate start of a transaction
splog(addr, value)
record *value* at address *addr*
tx_commit()
persists speculative log and commit
recover_from_splog()
post-crash recovery with splog

```
1 tx_begin() //tx #1
2   a = 1
3   splog(&a, 1)
4   b = 2
5   splog(&b, 2)
6   ...
7 tx_commit()
8
```

Figure 3: The SpecPMT API (left) and the example codelet (right)

updated by committed transactions. SpecPMT handles both, where its recovery revokes the first kind and handles the second kind by discarding all log records generated by uncommitted transactions and replaying all fresh log records. A log record is fresh if it records the updated committed value of a memory location. Effectively, the speculative logs function as redo log entries for completed transactions and undo log entries for failed transactions.

3.1 Speculative Logging Example

During normal execution, transactions generate speculative log records and update memory locations in persistent memory. Each thread manages its own log without consulting with other threads. A background log reclamation thread discovers and recycles log records useless for recovery.

We illustrate speculative logging with a running example in Figure 4, which shows two transactions that both update memory locations *a* and *b* (The example applies to both software and hardware SpecPMT proposed in this paper.). The persistent memory state for data and logs are shown for different snapshots in time. The snapshots begin when the first transaction has just committed. Here both locations have been updated, and speculative log records for *a* and *b* have been created. As the second transaction executes (second snapshot with *b=10*), it creates log records for *a* and *b*, and appends them in the log. Note that at this point, if a crash occurs, the first transaction log records are sufficient to restore data to the point before the second transaction by undoing any changes to *a* and *b* in the second transaction. Hence, new data values and the associated log records in the second transaction may remain in the volatile memory. When the second transaction commits (third snapshot), the commit ensures the new speculative log records persist along with the transaction commit metadata. Note that updates on locations (e.g., *a*) do not need to persist (e.g., be flushed) at this point, as second transaction log records are sufficient to replay the non-persisted data updates if a crash occurs (the speculative log entries function as a redo log for the just committed transaction). Finally, when log reclamation is triggered (last snapshot), explicitly or implicitly, the reclamator finds that log records from the first transaction are stale and thus can be removed. The first transaction metadata is also removed since no fresh log record remains.

The post-crash recovery is straightforward. Log records associated with an uncommitted transaction are discarded after checking the transaction commit metadata. The remaining fresh log records (from committed transactions) are then used to restore the values of

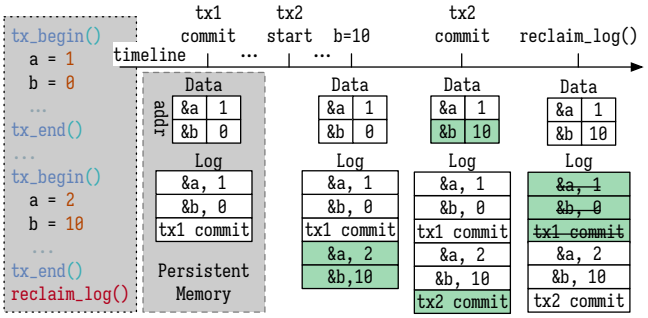

Figure 4: A codelet and the memory state snapshots with timeline, illustrating the mechanism of speculative logging

logged memory locations, effectively undoing interrupted transactions and redoing completed transactions. For example, after a crash interrupts the second transaction (second snapshot with *b=10*), the recovery discards the last two records, and replays the first two records to restore the values of memory locations (i.e., *a* and *b*).

4 SOFTWARE SPECULATIVE LOGGING

Implementation of software speculative logging centers on a sequential log record organization and log reclamation. Figure 5 illustrates the implementation at a high level. During normal execution, the transactions append log records to the log area and update the durable and volatile memory data. The background log reclamator uses a volatile record hash index to determine the log record freshness. The recovery routine uses log records to revoke uncommitted updates and replay the most recently committed updates.

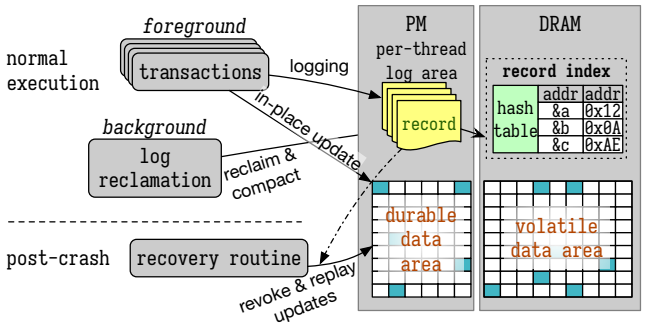

Figure 5: The building blocks of software SpecPMT

If a transaction updates a persistent memory location multiple times, the transaction only needs to log the last update because all other log records for the datum would be stale if created. Stale log records are useless for both normal execution and recovery, causing additional write traffic. Similar to existing undo and redo logging schemes, the first or last update on a datum in a transaction can be discovered via write-set indexing [24] or compiler assistance [6, 71].

Software SpecPMT organizes log records in a sequence with new records always appended to the sequence tail. Such a design produces good spatial locality (which enables fast log record creation), but wastes memory as multiple updates from multiple transactions to the same datum may result in multiple log records appended to the sequence. In such a case, the latest log record for the datum is fresh, while others are stale. To reduce memory waste, software SpecPMT needs a background log reclamation to remove and deallocate stale log records.

A more memory-space efficient alternative would be to set a limit of only one log record for each datum. To achieve that, a hash table indexed by each datum's address may be used to locate its most recent log record. On each data update, the design replaces a now-stale log record with a new one. Such a design conserves memory space but sacrifices spatial locality. Considering that persistent memory is much denser than DRAM but slower to write, especially with a random write pattern, a sequential log record is likely the better choice [78]. Our experiment confirms this, with the hash table approach incurring 3.2× slowdown over the sequential log design (methodology described in Section 7).

4.1 Log Organization

Because the log area contains both stale and fresh log records, its organization should encourage fast freshness checking to assist log reclamation.

Software SpecPMT organizes the per-thread log area logically as a sequence of records where new records can only be appended; hence it keeps a chronological order of records. On a crash recovery, data updates can be replayed starting from the oldest log records to the youngest. Some unreclaimed stale log records may still be present and get reapplied, but this is fine because, eventually, the stale updates will be replaced by fresh log records.

Because the log sequence grows with the durable data write set size and decreases depending on the frequency of log reclamations, it is difficult to retain the log records in a fix-sized log area. Therefore, the software SpecPMT implements a dynamically extendable log area, by allocating segments (called *log blocks*) of memory spaces on-demand. As illustrated in Figure 6, each thread-private log area contains multiple log blocks connected with block pointers. Each log block contains log records that contain transaction commits, a set of log entries, and a forward block pointer if it is the last log record in the log block.

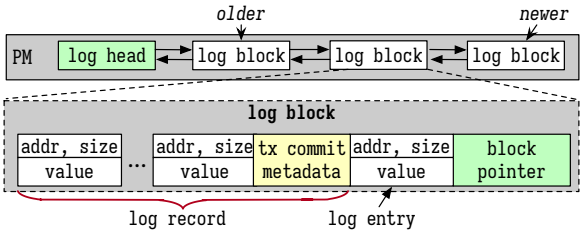

Figure 6: Log area organization

The transaction metadata contains the size of the log record and the checksum of the log record. If the software is multi-threaded, the metadata also contains a timestamp of the transaction commit. Each log entry includes the datum address, size, and value.

On transaction start, the software SpecPMT reserves memory space for transaction metadata. The transaction appends log records

to the log block during transaction execution. When a log block is full, the SpecPMT allocates a new log block and chains it to the log sequence with a block pointer.

On commit, the transaction sets the size of the log record and calculates the checksum for the entire log record, including the metadata, log entries, and block pointers. The checksum also serves as the transaction's commit status: the software considers the transaction as committed if the checksum matches the log record; the checksum is flushed and fenced to ensure persistence on transaction completion. Similar to a prior work [9], this design avoids a dedicated flag and a fence recording the commit status [66]. For multi-threaded software, the transaction sets the timestamp according to the hardware timestamp counter through *rdtscp* instruction on X86 processors (transactions are otherwise de-conflicted using a programmer specified locking-scheme — see Section 4.3.). The recovery uses the timestamp to determine the freshness of every log entry created by concurrent threads.

On a system crash, the recovery finds the first log block through a log head pointer. The pointer is located at a reserved memory location, such as the head of a persistent memory object pool [59]. The recovery then iterates over log entries and applies them to durable data. The recovery stops once a corrupted log record is encountered because there should not be fresh records afterward.

4.2 Background Log Reclamation

Log reclamation occurs in the background on a dedicated thread. Reclamation is triggered explicitly through an API or implicitly when a transaction execution finds the memory space overhead reaching a tunable threshold. Log reclamation reduces the memory consumption of the log records according to the durable data write set size of the software. Ideally, each updated durable datum is associated with only one log record. Our hardware solution introduced in Section 5 further allows the software to set arbitrary memory consumption bound.

During a log reclamation cycle, the reclamator scans backward starting from a log block, and traverses to the oldest log block along the chain of log blocks. Although this is the most effective when the scan begins with the youngest log block (i.e., at the tail), the scan may instead start from an arbitrary log block with potentially reduced effectiveness. Based on this observation, our log reclamation begins with the youngest *full* log block that no working thread is currently updating it, which eliminates the need for concurrency control.

Performance and crash consistency are important aspects of log reclamation. To maintain high performance, log reclamation uses a hash table while scanning the log records to determine the freshness of a given log record. Because the hash table maps addresses to entries, any resulting collisions quickly reveal potentially stale log records. The hash table itself does not require crash consistency, as it can be rebuilt when affected by a crash, and log reclamation can be repeated from the beginning if it is interrupted by a crash. From this observation, we choose to allocate the hash table in the volatile memory, which improves performance while allowing a *rebuild-on-crash* policy.

Performance is also optimized when the reclamator performs periodic *compacting*. For each time compacting cycle, the reclamator allocates new log blocks and copies fresh log entries from old log records into the new block, forming new compact log records in which the timestamp is set to the newest log entry. Upon completion, the reclamator inserts the new log blocks into the chain. The former step (copying) can be repeated after a crash. Thus, only the latter step (inserting the new log block) must be performed in a durable atomic manner. To achieve this without relying on a transaction, we first update the new log block's forward and backward pointers. We then update the predecessor log block's forward pointer to point to the new log block. Finally, we update the successor log block's backward pointer to point to the new log block. A crash during this insertion operation is recovered by using forward pointers to traverse the list and correct any backward pointers that did not persist. As a result, each log reclamation cycle requires only two fences — one to ensure the persistence of the new log block and one for the new log head pointer.

4.3 Programming Model

4.3.1 Switching Crash Consistency Mechanism. Software components may build upon other crash consistency mechanisms such as undo (i.e., PMDK) or redo logging transactions, which allows these mechanisms to coexist with speculative logging and increases practicality and compatibility.

To achieve this, SpecPMT allows switching from speculative logging to another crash consistency mechanism. Because SpecPMT uses in-place updates, it only needs to flush dirty cache lines of durable data at the transition point. Once completed, speculative logs are no longer needed for crash recovery, and the new model can be used from that point forward. The software should ensure that there is no running SpecPMT at the transition point. The flushing can be performed through entire-cache flushing, e.g., using instructions, such as *wbnoinvd*, or selective flushing through software analysis of record indices and *clwb*s.

4.3.2 External Data. When software updates durable data generated by other software or by other executions of the same software, there is likely no speculative log record associated with the data. This causes a consistency concern for SpecPMT. To rectify this, the software can update the external data in a crash-consistent manner by creating a snapshot prior to data modification. This is a common solution for checkpointing-based crash-consistent systems [9, 28, 48, 52] or out-of-place update persistent memory transactions [14, 46] that face similar issues when opening an existing persistent memory object pool [72–74, 79, 81]. Prior studies [9, 48, 52] have proposed an array of optimizations to alleviate the overhead generated by creating snapshots. Recent research [52] indicates that occasional checkpointing only adds 0.8%–10% execution overhead. SpecPMT only snapshots the data once, rather than periodically. Furthermore, it is possible to begin processing the external data with undo logging and checkpoints asynchronously. Once checkpointing finishes, the software can switch to speculative logging.

4.3.3 Concurrency Control. Like other persistent memory transactions [13, 59, 62, 63, 76], SpecPMT provides atomic durability and relies on the software to provide isolation in a multi-threaded context. The software can combine SpecPMT with concurrency

control mechanisms, including, but not limited to, optimistic concurrency control [46, 65] and strict two-phase locking [24, 62]. The speculative logging transactions must coincide with the outermost critical sections [42, 58].

5 HARDWARE SUPPORT

While the software-only design for speculative logging enables performant crash-atomicity, it suffers from multiple drawbacks: it nearly triples the memory space overhead, and it requires a core dedicated to running timely background log reclamation. The background log reclamation threads also require tuning the trigger level for reclamation and increasing the memory bandwidth pressure. To address these challenges, we propose hardware support for selective logging, which we will refer to as hardware SpecPMT.

Hardware SpecPMT institutes architectural changes to support primitives that (a) identify hot pages, (b) perform undo logging, and (c) perform bulk copying. These primitives support two novel roles: *undo-speculative hybrid logging* and *epoch-based log reclamation.*

5.1 Hybrid Logging

Hardware SpecPMT's hybrid logging combines fast but memory-consuming speculative logging for frequently updated (i.e., hot) data — which constitutes only a small portion of the software memory footprint — with slow but memory-saving undo logging for infrequently updated (i.e., cold) data.

Hardware SpecPMT relies on hardware support to distinguish between hot and cold data at the granularity of pages, based on the frequency of data updates on the page. A page may switch between hot and cold inside a transaction. Figure 7 illustrates how the logging switch is handled based on data hotness transition.

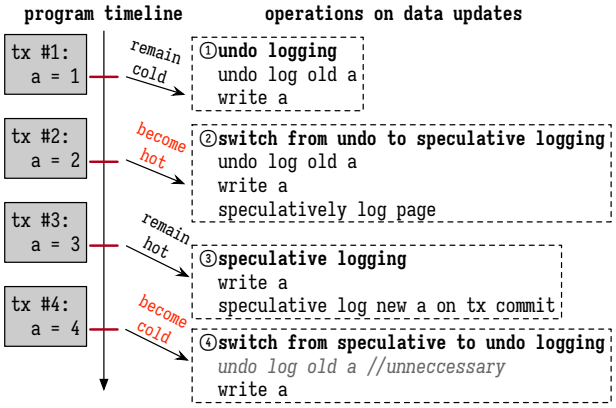

Figure 7: Illustration of logging switches based on changes in data hotness.

The figure illustrates four consecutive transactions updating the same data and alternating between cold and hot. The datum is initially identified as cold in the first transaction; hence undo logging is applied. In the second transaction, the datum becomes hot due to additional data updates, and the page switches from undo logging to speculative logging. In the third transaction, the datum is speculatively logged (at cache line granularity) because it remains hot. In the final transaction, the datum switches from hot to cold, and because it was already speculatively logged in the prior transaction, additional logging is not necessary. There must be a prior speculative log record for the datum that can serve as an undo log. However, for simplicity, hardware SpecPMT adopts the same logging strategy for a cold page.

In order to distinguish between hot and cold pages, we can associate metadata with the page table entry to record the hotness of each page. However, that requires modifying the page table and page fault handler parts of the OS. To avoid modifications to the OS, we associate additional metadata with each *translation look-aside buffer* (TLB) entry instead, as shown in Figure 8. This enables tracking only pages that are covered by the TLB. If a TLB entry is evicted or invalidated, we can no longer track the page, but such a page is likely no longer hot. The primary benefit of tracking hot pages through TLB entries is the lower and bounded memory overheads, as the memory consumption due to speculative logging depends on the number of pages it is applied to, and the TLB only covers a fixed and small subset of pages, the memory overhead from hybrid logging is much smaller and bounded.

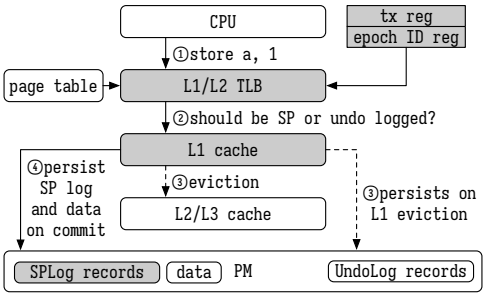

Figure 8: Overview of the hardware. Shaded components are extended.

Under hybrid logging, the TLB is checked on a write to data. On a TLB hit, the controller can determine whether the page is considered hot or cold. On a TLB miss, the controller treats such a page as cold. If the page is hot, it is speculatively logged, and a write to data updates the L1 cache directly. Otherwise, the hardware creates an undo log record for the cache line before updating it.

The details of the additional metadata for hotness tracking are shown in Figure 9. Each TLB entry contains a one-bit *EpochBit* and a three-bit saturating counter. If the *EpochBit* is set, indicating the page has been speculatively logged, the "cnt/EID" records the epoch ID for epoch-based log reclamation (to be discussed in Section 5.2). If the *EpochBit* is clear (indicating the page is cold), the counter records the number of transactional store operations on the page during the page residency in the TLB. When the counter reaches a threshold (for simplicity, the maximum value), the page is considered to have become hot. Such a page will start to be logged speculatively (i.e., by copying the entire page into the log). This transition is accomplished using a hardware bulk copying engine [39] (currently supported in ARMv9 [1]). During speculative logging, the hardware does not block access to the page as it still creates undo log records for the data before the store operation. After logging is completed, the

hardware sets *EpochBit* and sets the counter to the current epoch ID according to the epoch ID register.

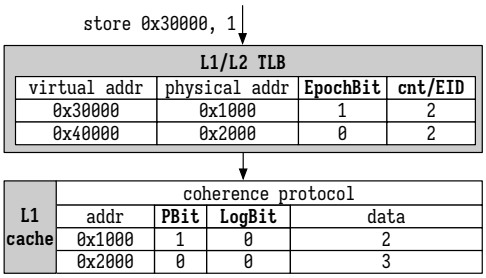

Figure 9: TLB and cache entries of hardware SpecPMT.

As discussed earlier, if the hardware evicts a TLB entry, the associated epoch ID and counter are discarded, and we treat the page as cold. As each core maintains its local epoch ID and hotness counter, hardware SpecPMT avoids costly synchronization [10] among TLBs on different cores.

Besides modifications to the TLB, hardware SpecPMT extends each L1 cache entry with two single-bit flags. The flags specify how the cache controller acts on a transaction commit or cache line eviction. *PBit* indicates whether the cache line needs persistence on eviction inside or outside transactions. The controller sets the bit when it updates the cache line of a hot page.

The cache controller sets *LogBit* after it undo logs the cache line or when it needs to speculatively log the cache line on transaction commit or cache eviction. For example, if both bits are set, the hardware needs to persist and speculatively log the cache line on transaction commit or eviction. If only *LogBit* is set, the cache line is undo logged in the transaction. The hardware clears the *LogBit* on transaction commit but reserves the *PBit*.

Invalidating speculatively logged cache lines on cache coherence events remains the same, which is a compelling property as modifying coherence protocol is error-prone and perhaps hurts performance. Once speculatively logged, the invalidated cache line requires no persistence. Consider MSI coherence, where a transactional write w_1 turns a cache line to M state and commits. A subsequent transaction on another core updates the same data to w_2. The hardware writes back the dirty cache line to the shared cache, followed by transiting the state from M to I. In the transactions, the hardware creates a speculative log record for w_1 and w_2, respectively. If the transaction encasing w_2 commits before a system crash, the recovery uses the newer log record to recover the most recent value of the cache line, i.e., w_2. If the crash interrupts the transaction, the recovery uses the log record for w_1 to revoke the effect of w_2. In both cases, the system does not have to persist the effect of w_1. Each core maintains its own speculative log records and records the timestamp for each transaction commit. The recovery first uses the timestamp to find the most recently committed speculative log record for a datum.

5.1.1 Correctness. Hybrid logging guarantees software recoverability as it ensures that each uncommitted transactional update is associated with a log record, either undo or speculative. Consider a cold page update in a crash-interrupted transaction. Whether the page becomes hot in the transaction or not, the hardware undo-logs the cache line before it performs the update. Regarding a hot page update, there are two cases. If the page was hot before the transaction started, there must be a speculative log record about the updated data. Otherwise, if the page becomes hot in the transaction, the hardware must speculatively log the page before setting the page as hot. The page log record serves as an undo log record for the subsequent updates in the transaction.

The protocol maintains two invariants: (1) all uncommitted undo log records are fresh; (2) all uncommitted page log records serve as undo logs for some data and speculative logs for other data, which must be undo logged prior to the creation of the page log records in the same transaction. Therefore, the recovery guarantees to revoke any uncommitted transactional update and preserve any committed update with three steps: (i) It applies the uncommitted speculative page log records to the data; (ii) It applies the uncommitted undo log records to the data; (iii) It applies committed speculative log records in chronological order.

5.1.2 Performance Guarantee. Although the hardware performs speculative logging at cache line granularity on hot pages, it performs speculative logging at a page-level granularity when the page switches from cold to hot. Hence, there is some possibility that speculative logging may increase the latency of execution as well as write traffic to persistent memory if there are only a few updates on the page. Several aspects may reduce this possibility. First, hardware SpecPMT logs only hot pages, which tend to be written many times in a period. Second, speculative logging defers flushing to transaction commit, allowing write coalescing of logs within a transaction. It also defers data persistency to cache line eviction, allowing write coalescing of data across transactions. Section 7 quantifies the effect of hardware SpecPMT on write traffic and memory consumption. Third, the hardware may provide an API to enable/disable speculative logging, which sets/resets a control status register bit. This allows the programmer or user to disable speculative logging (and rely solely on undo logging) if it produces an adverse performance impact. Finally, it is possible for hardware SpecPMT to sample the performance of undo logging and speculative logging for a frequently executed transaction to compare and choose the logging scheme that performs better than the other.

5.2 Epoch-Based Log Reclamation

Another advantage of hardware SpecPMT over a software-only solution is that its method of performing reclamation is epoch-based, fast, in the foreground, and thread-local. Thread-local reclamation has been a long-time goal for redo logging optimization [13–15, 25, 26, 46]. The transaction execution model must apply the freshest redo log record to the data, often requiring synchronization or substantial hardware modifications. Unlike redo logging, speculative logging finds stale log records locally without the need for thread synchronization.

Hardware SpecPMT epoch-based log reclamation is a software-hardware co-design that provides two key benefits: (a) it allows the software to set arbitrary memory consumption limits by reclaiming hybrid log records by epochs, and (b) it allows each thread to reclaim

its thread-local speculative log records without consulting other threads.

Log reclamation in hardware SpecPMT is achieved by dividing a thread execution into epochs. Each private TLB entry is augmented with an epoch ID to record the epoch in which the page was speculatively logged. In a log reclamation cycle, the software reclaims all speculative log records created inside an epoch by clearing the epoch ID in TLB. The core idea of epoch-based log reclamation is to switch a set of hot pages with the same epoch ID into cold pages with only one instruction. The hardware clears the epoch bit and the EID field of each TLB entry to set the page as cold. Two new instructions are added, *clearepoch EID* and *startepoch EID*, to end and start an epoch with a given epoch ID, respectively. As long as the software always clears the oldest epoch, it reclaims the log records at the beginning of the log area because the log records are chained in chronological order.

When a transaction commits, the hardware scans the L1 cache to find dirty cache lines updated by the transaction. It creates and persists log records for the speculatively logged pages and cache lines. It skips the persistence of those updated cache lines. It persists the undo logged cache lines.

Hardware SpecPMT allows an L1 cache line updated in the transaction to overflow to the L2 cache as long as the hardware speculatively logs the cache line prior to the eviction. This allows a large transaction to succeed.

The reclamation incorporates a set of data structures and hardware components shown in Figure 10, including the log records, eight epoch pointers in DRAM, the hardware maintaining the pointers and registers, and two new instructions starting and reclaiming an epoch. The log records are logically grouped by epochs but physically stored in log blocks. The epoch pointers refer to the head of the log record of each epoch log record group.

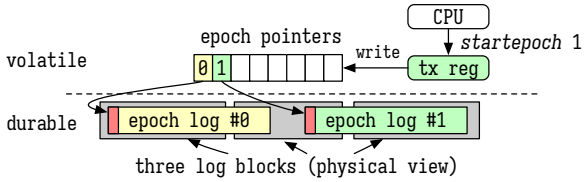

Figure 10: The core components of the epoch-based log reclamation

5.2.1 Log Reclamation for Sequential Software. During normal execution, hardware SpecPMT software checks whether or not to reclaim log records after each transaction commit. The checks can be optional and adaptive because log reclamation is only needed occasionally. The software is responsible for determining which epoch to reclaim. This work adopts a straightforward strategy that always reclaims the oldest epoch.

The software performs log reclamation in three steps. Similar to handling a transaction mode switch (Section 4.3), it first persists all speculatively logged data in the current epoch. This can be achieved through scanning the log record and selectively flushing data addresses indicated in the log records via *clwb*, or writing back the entire L1 cache through instructions such as *wbnoinvd*.

After the first step is completed, in the second step, the software invokes a new unprivileged instruction, *clearepoch EID*, to switch some pages from hot to cold by clearing the *EpochBit* and the cnt/EID field for pages that match the *EID*. Those pages were initially speculatively logged in an epoch numbered as *EID*. In the third step, the software reclaims the memory space occupied by the log records of the epoch. The software performs the second and third steps without ordering constraints.

It is possible to merge the first and the second step by extending *clearepoch* with the L1 cache scanning and data persistence semantic. This design also enables optimization to avoid excessive data persistence if a page was logged in an old epoch, but some of its cache lines are also logged in new epochs. The hardware can skip persisting those cache lines when it reclaims the old epoch. The optimization incurs three extra bits for each L1 cache line.

The software starts a new epoch with a new unprivileged instruction, *startepoch EID*, where *EID* refers to the ID of the new epoch. The instruction assigns the epoch ID register to *EID*. Epoch ID 0 is reserved for cold pages.

Epoch size selection involves a trade-off. Small epochs cause excessive log reclamation, and TLB and cache flushes, while large epochs consume more memory space. In current implementation, we start a new epoch when the epoch contains over 2MB of records or 200 speculatively logged pages. The first threshold bounds memory consumption, while the second threshold distributes speculatively logged pages into different epochs to improve performance.

5.2.2 Log Reclamation for Multi-threaded Software. Multi-threaded software introduces challenges in correctness and scalability. Figure 11 illustrates an example involving two threads performing writes to the same data, with transactions shown in black boxes. The figure shows that reclaiming an epoch of thread can cause data corruption in a crash. Specifically, when the second thread reclaims its epoch, it removes the speculative log record for $w2$. Later, when another thread updates the data (w_3) but suffers from a system crash before the transaction commits. In this case, the recovery cannot revoke the update as the data cannot be recovered to the state after $w2$.

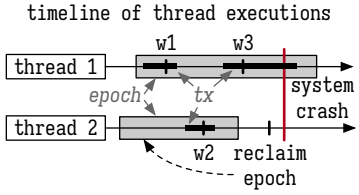

Figure 11: Illustrating how log reclamation may prevent crash recovery from revoking w_3. The three writes update the same memory location, and solid black boxes represent transactions.

To address this, we propose a simple non-blocking reclamation protocol. We let an epoch be inactive if its epoch ID has been reassigned to a younger epoch of the same thread. Formally, the software can safely reclaim all log records in an epoch e if: (1) e is an inactive epoch; (2) all active epochs must start after the end of e, including the epochs belonging to other threads.

The protocol preserves recoverability. Consider a speculatively logged datum *a* inside an epoch *e*. Because *e* is inactive, before the software reassigns the epoch ID to a younger epoch, the software must clear the *EpochBit* for all speculatively logged pages in *e*, including the page containing *a*. If the software updates *a* in a closed active epoch, which means the epoch has been ended, but its epoch ID has not been reassigned, then the log record for *a* created in *e* is stale and safe for reclamation. If the software updates *a* in an opening epoch because the software has recycled every epoch starting before the end of *e*, then the software must consider the page of *a* as a cold page. Then the software must undo-log *a* before updating it.

Hardware SpecPMT realizes the protocol by letting each thread maintain a timestamp of when the earliest unreclaimed epoch starts. On reclaiming an epoch, the software checks the timestamps and the activeness of the associated epochs of all threads. If active epochs overlap with the epoch to reclaim, the software defers the check and log reclamation to further transaction starts or commits.

The recovery of hardware SpecPMT is the same as software SpecPMT. It scans every thread's log records and replays them according to the timestamp.

5.3 Other Issues

5.3.1 Persistent Caches. Some processors may provide persistent or battery-backed cache hierarchy such as eADR [59]. The processors may run transactions without logging any data if it retains the entire write set of the transaction in the cache and provides a mechanism to identify and discard uncommitted data update retained in the cache. However, the adoption of eADR may be limited [59] due to costly hardware and maintenance [8]. The costs further increase with larger caches and larger systems (e.g., NUMA).

5.3.2 Transaction Abort. A persistent memory transaction may abort on an application or system exceptions, such as running out of memory space. Whereas the software can always revoke the effect of the interrupted transaction via the slow crash-recovery routine, it can exploit the transaction abort mechanism of transactional memory [55, 82] to enable fast abort during normal execution, such as preserving the write set in a private cache and discarding all uncommitted updates on abort.

5.4 Hardware Cost

Hardware SpecPMT incurs 0.91KB on-chip storage. It adds two bits to each L1- and L2-TLB entry; and two bits to each L1 data cache entry. For Skylake micro-architecture, the L1 data cache, L1-, and L2-TLB contain 512, 64, and 1,536 entries, respectively. The hardware also devotes two registers to retain the transaction state and current epoch ID. Together, hardware SpecPMT incurs less than 0.04% on-chip storage for a Skylake core.

6 DISCUSSION

Alternative Uses. Speculative logging may augment durable write-ahead logging transactions despite the storage they use, such as SSD or remote memory. Unlike typical implementations of write-ahead logging that underpins database systems [64], file systems [38, 85], and distributed storage systems [7, 86], speculative logging does not require a software cache (such as buffer pools managed by a database) to retain data changes but still allows the log records and data changes generated by a transaction to reach persistent domain in any order. Therefore, speculative logging can potentially reduce transaction commit latency, improve throughput, and reduce write traffic for systems where write-ahead logging plays a key role in consistency, crash recovery, and performance.

Alternative Designs. Speculative logging transactions are not bound to specific implementations such as the design proposed in this paper. For example, a design may offload the hotness checking to software. The software may use a performance monitoring unit [84] or manipulate page table entries [21, 77, 80] to sample or count the accesses to pages without modifying the hardware. The hardware performs only epoch controlling.

Compiler Optimization. SpecPMT may use a compiler to reduce memory consumption in maintaining the log records. Given that a compiler can sometimes infer what data a transaction will update and when they are updated, the software can discard log records and then rebuild them at an appropriate point. For example, the software can skip log record creation in the first speculative logging transaction shown in Figure 2 and create a log record for the data before the second transaction starts.

7 EVALUATION

This section evaluates the performance of the proposed solutions. It provides root cause analysis of the performance gain, including a reduction in write traffic and the number of fences.

7.1 Methodology

7.1.1 Benchmarks. We evaluate the software and hardware speculative logging with all transactional applications from STAMP [53] except for *bayes* due to its unstable performance [14, 16, 25]. STAMP has been used for many persistent memory transaction studies [14, 25, 41, 71]. We port the transactional applications to persistent memory with libvmmalloc [3, 81], which overrides dynamic memory allocation to persistent memory allocation. We evaluate the software solution with non-simulator input of STAMP and the hardware solution with simulator workload. The *kmeans* and *vacation* applications have two kinds of workload: low and high contention. We use both. Other applications have only one workload.

7.1.2 Software Platform. We evaluate the software solution on an Intel Gold 6230 machine equipped with 1TB@2,666MT/s (eight DIMMs) first-generation[1] Intel Optane persistent memory and 128GB@2,666MT/s DDR4 memory. We mount persistent memory with a DAX-enabled file system.

We compare the software solution, software SpecPMT (SpecSPMT), with PMDK and two recent in-place update persistent memory transactions, plus a sub-optimal implementation of software SpecPMT:

- PMDK [59]: Intel's industry-standard persistent memory programming transaction mixing undo and redo logging. We use it as the baseline.

[1]The second-generation Optane persistent memory DIMM runs at largely the same speed at lower power budget [11].

- Kamino-Tx [51]: a state-of-the-art in-place data update transaction. Our implementation omits the data copying from the main copy to the backup copy. Therefore, our experiments correspond to Kamino-Tx's upper bound in performance. The implementation logs every write intent's address.
- SPHT [14]: a state-of-the-art redo logging transaction that works on a volatile data snapshot and replays the log to the persistent data with background threads. We use its forward linking version and a background log replayer thread.
- SpecSPMT-DP: sub-optimal software SpecPMT with enforced data persistence on transaction commit. We use it to measure the gain from removing the fences and the data persistence.

7.1.3 Hardware Simulation. We built the simulator on an x86 core model for convenience, but the design is not bound to a specific instruction set architecture. Our simulator is built on top of the system-level Gem5 simulator [12]. We evaluate the modified cache with the Gem5 integrated memory subsystem simulator, Ruby. Table 1 lists the parameters of the simulated hardware.

Table 1: System configuration

Component	Parameter
CPU	out-of-order X86 core@4GHz, MESI cache coherence protocol
L1 TLB	Private per core, 64 entries, 8-way
L2 TLB	Private per core, 1536 entries, 12-way
Data Cache	Private per core, 32KB, 8-way, 2 cycles
L2 Cache	Shared 2MB, 12-way, 20 cycles
DRAM	DDR4 2400Mhz, tRCD/tCL/tRP/-tRAS/tWR=14/14/14/32/15ns
PM	512 bytes write pending queue, 10ns; 150ns read latency; 500ns write latency

We compare hardware SpecPMT (SpecHPMT) with two persistent memory transactions and two sub-optimal designs.

- EDE [63]: a state-of-the-art in-place update transaction. It eliminates fences between logging and data update operations. We coalesce the log records as much as possible. We use it as the baseline.
- HOOP [13]: a state-of-the-art out-of-place update transaction. It removes fences and enables asynchronous data persistence. It requires 273KB of dedicated on-chip storage per core and an additional core to run background *garbage collection* (GC). We ignore the latency on address redirection to model performance optimistically. We optimize the GC by coalescing log records before applying them to the data. The GC reclaims 128KB log records at each GC cycle to avoid excessive contention on the 16KB on-chip eviction buffer.
- SpecHPMT-DP: hardware SpecPMT with data persistence on transaction commit. This is a suboptimal variant; we use it to measure the gain from removing data persistence.
- no-log: Transactions without logging. It persists data on transaction commit. Its performance is ideal for in-place update persistent memory transactions. It does not provide crash consistency.

7.2 Software Solution Evaluation

SpecSPMT-DP and SpecSPMT consistently outperform PMDK and Kamino-Tx, as shown in Figure 12. On average, SpecSPMT-DP achieves 3× and 1.78× speedups over PMDK and Kamino-Tx, respectively. SpecSPMT improves the performance by removing data persistence. It achieves 5.1× and 3.02× speedup over PMDK and Kamino-Tx, respectively.

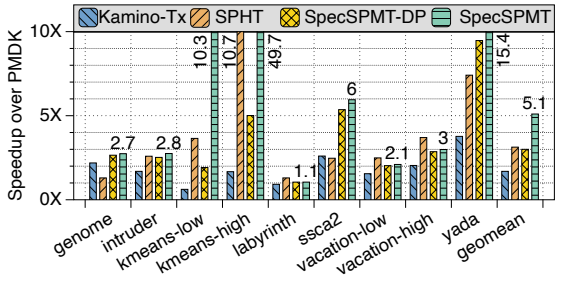

Figure 12: Speedup over PMDK. Evaluated on a real machine

SpecSPMT-DP considerably outperforms Kamino-Tx when an application updates a large amount of data because SpecSPMT-DP removes all fences after each update. For example, SpecSPMT-DP is at least 1.5× faster than Kamino-Tx on the five applications with the largest number of transactional updates, as shown in Table 2. We classify the five applications as write-intensive applications. Among the remaining four applications, which we classify as write-moderate applications, SpecSPMT-DP is less than 1.4× faster than Kamino-Tx.

Table 2: Size and number of transactions

Application	Avg. size (B)	Num of tx	Num of updates
genome	7.2	2,489,218	7,230,727
intruder	20.5	23,428,126	106,976,163
kmeans-low	101	9,874,166	266,600,674
kmeans-high	101	4,106,954	110,887,006
labyrinth	1420	1,026	184,190
ssca2	16	22,362,279	89,449,114
vacation-low	44.2	4,194,304	31,582,272
vacation-high	67.8	4,194,304	43,950,938
yada	175.6	2,415,298	57,844,629

SPHT [14] offloads the data persistence to a background replayer thread. On the critical path of transaction commit, it persists only redo log records. By removing the fences on logging operations, SPHT outperforms Kamino-Tx with a similar speedup to SpecSPMT-DP.

As SpecSPMT gains from removing mandatory data persistence, it considerably outperforms SpecSPMT-DP by up to 9.9× on the write-intensive applications. However, on the write-moderate applications, SpecSPMT is only up to 5% faster than SpecSPMT-DP. It gains noticeably higher speedup among the write-intensive applications with large transaction sizes. Specifically, on the two versions of *kmeans* and *yada*, of which the average transaction size is larger than 20 bytes, SpecSPMT achieves 9.9×, 5.4×, and 1.6× speedup.

The other two write-intensive applications, *intruder* and *ssca2*, have average small transactions with four bytes write-set. SpecSPMT brings about a 10% speedup by removing the data persistence.

7.3 Hardware Solution Evaluation

SpecHPMT outperforms the baseline EDE by 1.41× on average, as shown in Figure 13. SpecHPMT achieves substantial speedup on write-intensive applications like the software solution, except for *kmeans-low*. This application devotes much time to computation between consecutive transactions, leaving the hardware enough time to drain the write pending queue before the next transaction starts. The *kmeans-high* has less computation and therefore observes higher speedup as durable data update is the bottleneck.

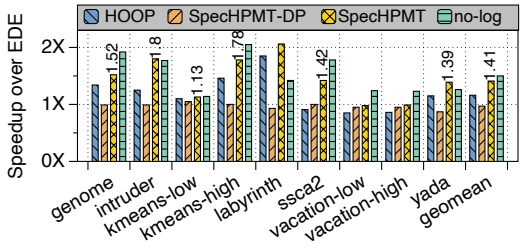

Figure 13: Speedup over EDE. Evaluated with simulator hardware

On average, HOOP is 1.19× faster than EDE as it moves the data persistence off the critical path of transaction commit. It also reduces memory traffic by 18.9% times by coalescing the log records from multiple transactions.

While HOOP reduces the critical path latency, its occasional garbage collection exhausts the write buffers on the memory controller, causing intensive write contention with application working threads. SpecHPMT avoids such contention by allowing speculatively logged data to stay in the cache or naturally overflow to persistent memory on cache eviction. Therefore, it outperforms HOOP by 1.21× on average, even despite the fact that HOOP requires an additional core and more than 200× the on-chip storage.

SpecHPMT is only 0.09× behind the ideal case (i.e., 1.5× speedup vs. 1.41× with SpecHPMT), *no-log*. On *labyrinth* and *yada*, SpecHPMT even outperforms *no-log* as it replaces distributed persistent memory writes with sequential log writes, which is faster on persistent memory [11].

Eliminating the ordering between log and update operations brings about marginal speedup as the SpecHPMT-DP performs nearly the same as EDE. They cause largely the same amount of write traffic on both data and log persistence, as shown in Figure 14. The only difference is that EDE maintains the ordering between the log and data update operations, while SpecHPMT-DP does not. However, the out-of-order core hides the overhead of ordering.

SpecHPMT delivers the second-lowest write traffic among all the designs. EDE and SpecHPMT-DP incur the most write traffic among all designs. Whereas HOOP also persists log and data, unlike EDE and SpecHPMT-DP, which coalesce data persistence by individual transactions, HOOP coalesces data persistence across transactions. If multiple transactions log the same datum multiple times, the GC coalesces the log records and applies only the latest record to

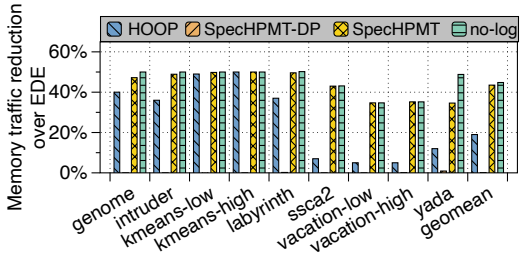

Figure 14: Reduction of write traffic. Higher is better.

the data. Consequently, HOOP achieves the write traffic as low as SpecHPMT on half of the applications. Furthermore, HOOP creates a log record for each data update and cache miss in a transaction. Therefore, it produces excessive logs on the applications (*ssca2, vacation, yada*) with large memory footprints twice the average of remaining applications. Comparatively, SpecHPMT creates log records only for data updates.

7.3.1 Memory Consumption. SpecHPMT gains higher speedup if the software devotes more memory space to the log area, which implies that the hardware speculatively logs more data. Unlike HOOP, the size of the on-chip mapping table bounds the log size; the log area of SpecHPMT can grow unboundedly. We varied the epoch size to analyze the sensitivity to memory consumption. Figure 15 shows the average speedup and write traffic reduction on the average memory consumption. When the benchmarks tolerate 15% and 20% additional memory consumption, they achieve 1.36× and 1.4× speedups, respectively. Even when the memory consumption is small (2.6%), the speedup is still substantial (1.12×).

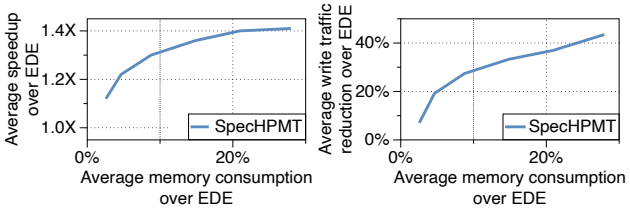

Figure 15: Average speedup and write traffic reduction various on average memory space increments

Small epochs hurt performance as the software logs a page and reclaims the log record before updating any data on the page. *vacation* observes 26% to 8% performance degradation when the memory consumption varies from 2.6% to 14.6%. Section 5.1 describes an optimization that reverts the system to undo logging to avoid such unnecessary performance loss.

8 RELATED WORK

A number of studies attempted to reduce the overhead of persistent transactions, as summarized in Table 3. We classify them into three groups as follows.

Asynchronous Data Persistence. A class of techniques hide the latency of data persistence by writing back durable data after transaction commit. Among them, Kamino-Tx and its variant [32, 51] are perhaps the most relevant to this work. They do in-place data

Table 3: Summary of related work

system	platform	ordering logging and data update	persistence domain	data persistence	data access	applicability
EDE[63]	hardware	non-fence ordering	unmodified	synchronous	direct	general
ATOM[42],Proteus[62]	hardware	non-fence ordering	modified	synchronous	direct	general
TSOPER[22],ASAP[5, 76]	hardware	non-fence ordering	modified	asynchronous	direct	general
HOOP[13],ReDu [37]	hardware	eliminated	unmodified	asynchronous	indirect	general
PMDK [59]	software	fence	unmodified	synchronous	direct	general
Kamino-Tx[51]	software	fence	unmodified	asynchronous	direct	general
Lsnvmm[31]	software	eliminated	unmodified	eliminated	indirect	general
Pronto[52]	software	eliminated	unmodified	eliminated	direct	data structure
SpecPMT (this work)	both	eliminated	unmodified	eliminated	direct	general

updates while keeping asynchronous data persistence. To achieve that, they maintain a backup copy of the data and a background thread asynchronously applies the updates from the main copy to the backup. On a crash, Kamino-Tx recovers the corrupted data from the backup copy. To identify which data are corrupted, Kamino-Tx records the addresses of all transactionally-updated data. Unlike SpecPMT, Kamino-Tx does not avoid the fences for ensuring address persistence before a main-copy data update.

Redo logging [13, 37, 46, 66] transaction techniques realize asynchronous data persistence via out-of-place data updates. They use background threads to apply the log to the data without blocking the transaction commit. They require additional address translation for every memory access, which causes additional memory accesses and concurrency control issues. Recent research explores hardware acceleration [13] for address translation, however it requires a large on-chip buffer to retain logged data (256KB per core). SpecPMT avoids address translation as it relies on in-place updates, and hence only requires 1KB on-chip storage per core, two orders of magnitude less than HOOP [13].

The dominant logging and recovery approach for disk-based database management systems, ARIES [54], achieves asynchronous data persistence and in-place data accesses by combining undo and redo logging. A prior work [57] enables undo+redo logging for persistent memory systems. Still, an undo+redo logging record must reach the persistent domain before the associated write intent. Thus, it incurs more persistent memory write traffic than speculative logging because it records both the old and new values of a memory location.

Recent efforts [5, 22, 76] exploit buffered persistence to enable both asynchronous data persistence and direct memory access. Those transactions assume that software can tolerate the loss of committed transactions on a crash.

Eliminating Data Persistence. LSNVMM [31]'s log-structured design retains every data update with a log record and appends it to a log area. The solution incurs considerable overheads on address translation as it redirects every data access to the most recent corresponding log with a tree. HOOP [13] outperforms it by 28%.

Pronto [52] exploits semantic logging that periodically checkpoints a data structure and records the operations on the data structure. It recovers a crash-corrupted data structure by re-executing the operations from a checkpoint. Other re-execution transactions exploit idempotence [33, 47, 71]. They are applicable for general programs but need to log all data necessary for re-execution, including volatile data and related registers.

Transactions With Reduced Fences. Early studies to reduce the use of fences [29, 42, 62] introduced additional components in the persistent domain. Recent works started eliminating the fence with no-fence ordering among instructions. Themis [60] implements immediate persistency. It enforced persist ordering between non-temporal and normal stores as it assumes the transaction persists log with non-temporal stores vs. data with normal stores. Pmem-spec [36] speculatively sends all persistent memory writes to a memory controller and the cache hierarchy. It causes a stale read problem, in which pmem-spec considers a virtual power failure and invokes a costly crash recovery. A recent work [63] allows arbitrary instruction persistence ordering by tracking the programmer-annotated instruction dependency in a modified write queue. All the solutions need to maintain ordering between logging and data updates, while SpecPMT does not.

Another branch of work [14, 25, 69] improves the concurrency of persistent memory transactions without paying attention to performance issues caused by fences and data persistence. Although speculative logging encourages concurrency by design, concurrency control is beyond the scope of this paper.

Log Reclamation. Log reclamation is a common issue for redo logging and multi-version concurrency control transactions [45, 56]. SSP [56] maintains a second physical page for each atomically updated virtual page. It consolidates the two pages when the associated page table entry overflows from TLB. Excite-VM [45] realizes snapshot isolation through an in-memory software cache for updated pages. Hardware SpecPMT requires neither a snapshot nor a software cache. Other hardware accelerations for general garbage collection [34, 49, 75] address the complexities of general applications, such as concurrency control or crash recoverability. SpecPMT removes all the mentioned complexities with log sequence design.

9 CONCLUSION

In this paper, we presented speculative logging, a new logging approach which removes most in-transaction fences and data persistence, enforces immediate persistence, and performs direct memory loads and in-place data updates. We discussed a software-only and a hardware-supported design for speculative logging. The former achieves a low 10% execution time overhead, compared to a state-of-the-art solution of 232%. The latter keeps the performance of the software-only solution while bounding its memory consumption. Compared to the state-of-the-art undo and redo logging, it lowers execution time overheads by 86% and 76%, respectively, while requiring a modest 0.91KB on-chip storage overhead.

ACKNOWLEDGEMENT

We thank anonymous ASPLOS shepherd, reviewers, and Alex Freij for their constructive feedback. This work is supported by the National Key Research and Development Program of China under grant No.2022YFB4500303, the National Natural Science Foundation of China under grant No.62202184 and No.61825202, and the National Science Foundation (NSF) under Grants CNS-2107068, 1900724, and 2106629. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] 2021. ARMv9 CPY Instructions. https://developer.arm.com/documentation/ddi0602/2021-12/Base-Instructions/CPYP--CPYM--CPYE--Memory-Copy-.
[2] 2022. Direct File Access. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.
[3] 2022. Intel libvmmalloc. https://pmem.io/vmem/libvmmalloc/.
[4] 2022. Persistent Memory – A New Hope. https://www.sigarch.org/persistent-memory-a-new-hope/.
[5] Ahmed Abulila, Izzat El Hajj, Myoungsoo Jung, and Nam Sung Kim. 2022. ASAP: Architecture Support for Asynchronous Persistence. In *Proceedings of the ACM/IEEE 49th Annual International Symposium on Computer Architecture*. 306–319.
[6] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. 2006. Compiler and runtime support for efficient software transactional memory. *ACM SIGPLAN Notices* 41, 6 (2006), 26–37.
[7] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 353–369.
[8] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. Bbb: Simplifying persistent programming using battery-backed buffers. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 111–124.
[9] Mohammad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy persistency: A high-performing and write-efficient software persistency technique. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*. 439–451.
[10] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the USENIX Annual Technical Conference*. 27–39.
[11] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-bench: benchmarking persistent memory access. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2463–2476.
[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.
[13] Miao Cai, Chance C. Coats, and Jian Huang. 2020. Hoop: efficient hardware-assisted out-of-place update for non-volatile memory. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. 584–596.
[14] Daniel Castro, Alexandro Baldassin, João Barreto, and Paolo Romano. 2021. SPHT: Scalable Persistent Hardware Transactions. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. 155–169.
[15] Daniel Castro, Paolo Romano, and João Barreto. 2019. Hardware transactional memory meets memory persistency. *J. Parallel and Distrib. Comput.* 130 (2019), 63–79.
[16] Daniel Castro, Paolo Romano, and João Barreto. 2019. Hardware transactional memory meets memory persistency. *J. Parallel and Distrib. Comput.* 130 (2019), 63–79.
[17] Cheng Chen, Jun Yang, Mian Lu, Taize Wang, Zhao Zheng, Yuqiang Chen, Wenyuan Dai, Bingsheng He, Weng-Fai Wong, Guoan Wu, Yuping Zhao, and Andy Rudoff. 2021. Optimizing in-memory database engine for AI-powered on-line decision augmentation using persistent memory. *Proceedings of the VLDB Endowment* 14, 5 (2021), 799–812.
[18] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. 81–95.
[19] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 133–146.
[20] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for OLAP workloads. In *Proceedings of the International Conference on Management of Data*. 339–351.
[21] Zhuohui Duan, Haikun Liu, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Yu Zhang. 2019. HiNUMA: NUMA-aware data placement and migration in hybrid memory systems. In *Proceedings of the IEEE 37th International Conference on Computer Design*. 367–375.
[22] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas, and Stefanos Kaxiras. 2021. TSOPER: Efficient coherence-based strict persistency. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 125–138.
[23] Hussein Elnawawy, Mohammad Alshboul, James Tuck, and Yan Solihin. 2017. Efficient checkpointing of loop-based codes for non-volatile main memory. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*. 318–329.
[24] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 237–246.
[25] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 59–74.
[26] Ellis Giles, Kshitij Doshi, and Peter Varman. 2017. Continuous checkpointing of HTM transactions in NVM. *ACM SIGPLAN Notices* 52, 9 (2017), 70–81.
[27] Derrick Greenspan, Naveed Ul Mustafa, Zoran Kolega, Mark Heinrich, and Yan Solihin. 2022. Improving the Security and Programmability of Persistent Memory Objects. In *Proceedings of the IEEE International Symposium on Secure and Private Execution Environment Design*. 157–168.
[28] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software persistent memory. In *Proceedings of the USENIX Annual Technical Conference*. 319–331.
[29] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. 2019. Distributed logless atomic durability with persistent memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 466–478.
[30] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally ordered durable data structures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 775–788.
[31] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *Proceedings of the USENIX Annual Technical Conference*. 703–717.
[32] Kaixin Huang, Sumin Li, Linpeng Huang, Kian-Lee Tan, and Hong Mei. 2020. Lewat: a lightweight, efficient, and wear-aware transactional persistent memory system. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 649–664.
[33] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 427–442.
[34] Jaeyoung Jang, Jun Heo, Yejin Lee, Jaeyeon Won, Seonghak Kim, Sung Jun Jung, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2019. Charon: Specialized near-memory processing architecture for clearing dead objects in memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 726–739.
[35] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded hardware transactional memory for a hybrid DRAM/NVM memory system. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 525–538.
[36] Jungi Jeong and Changhee Jung. 2021. Pmem-spec: Persistent memory speculation. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*.
[37] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 520–532.
[38] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Proceedings of the USENIX Annual Technical Conference*. 309–320.
[39] Xiaowei Jiang, Yan Solihin, Li Zhao, and Ravishankar Iyer. 2009. Architecture support for improving bulk memory copying and initialization performance. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*. 169–180.
[40] Hai Jin, Shuo Wei, Yan Sha, Chencheng Ye, Haikun Liu, and Xiaofei Liao. 2022. PMLiteDB: Streamlining Access Paths for High-Performance Persistent Memory Document Database Systems. *IEEE Trans. Comput.* (2022).

[41] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*. 660–671.

[42] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic durability in non-volatile memory through hardware logging. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 361–372.

[43] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 45–51.

[44] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. 2017. Viyojit: Decoupling battery and DRAM capacities for battery-backed DRAM. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 613–626.

[45] Heiner Litz, Benjamin Braun, and David Cheriton. 2016. EXCITE-VM: Extending the virtual memory system to support snapshot isolation transactions. In *Proceedings of the International Conference on Parallel Architectures and Compilation*. 401–412.

[46] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 329–343.

[47] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 258–270.

[48] Youyou Lu, Jiwu Shu, and Long Sun. 2016. Blurred persistence: Efficient transactions in persistent memory. *ACM Transactions on Storage* 12, 1 (2016), 1–29.

[49] Martin Maas, Krste Asanović, and John Kubiatowicz. 2018. A hardware accelerator for tracing garbage collection. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture*. 138–151.

[50] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. 2018. Persistent memory transactions. *arXiv preprint arXiv:1804.00701* (2018).

[51] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*. 499–512.

[52] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 789–806.

[53] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 35–46.

[54] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems* 17, 1 (1992), 94–162.

[55] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. 2006. LogTM: Log-based transactional memory. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 254–265.

[56] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L. Miller. 2019. SSP: Eliminating redundant writes in failure-atomic NVRAMs via shadow sub-paging. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 836–848.

[57] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 336–349.

[58] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture*. 265–276.

[59] Steve Scargall. 2020. *Programming persistent memory: A comprehensive guide for developers*. Springer Nature.

[60] Sara Mahdizadeh Shahri, Seyed Armin Vakil Ghahani, and Aasheesh Kolli. 2020. (almost) Fence-less persist ordering. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 539–554.

[61] Ali Sheikholeslami and P. Glenn Gulak. 2000. A survey of circuit innovations in ferroelectric random-access memories. *Proc. IEEE* 88, 5 (2000), 667–689.

[62] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 178–190.

[63] Thomas Shull, Ilias Vougioukas, Nikos Nikoleris, Wendy Elsasser, and Josep Torrellas. 2021. Execution Dependence Extension (EDE): ISA Support for Eliminating Fences. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture*. 456–469.

[64] Abraham Silberschatz, Henry F. Korth, and Shashank Sudarshan. 2002. *Database system concepts (seventh edition)*. Vol. 7. McGraw-Hill New York. 928–929 pages.

[65] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 18–32.

[66] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 91–104.

[67] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 496–508.

[68] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An early evaluation of Intel's optane DC persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–19.

[69] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. 2021. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. 141–153.

[70] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of Intel optane persistent memory: a close look at its on-DIMM buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 488–505.

[71] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: log less, re-execute more. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 346–359.

[72] Yuanchao Xu, Yan Solihin, and Xipeng Shen. 2020. Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 987–1000.

[73] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. 2022. Temporal Exposure Reduction Protection for Persistent Memory. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 908–924.

[74] Yuanchao Xu, Chencheng Ye, Yan Solihin, and Xipeng Shen. 2020. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. 680–692.

[75] Yuanchao Xu, Chencheng Ye, Yan Solihin, and Xipeng Shen. 2022. FFCCD: fence-free crash-consistent concurrent defragmentation for persistent memory. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 274–288.

[76] Sujay Yadalam, Nisarg Shah, Xiangyao Yu, and Michael Swift. 2022. ASAP: A Speculative Approach to Persistence. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 892–907.

[77] Dang Yang, Haikun Liu, Hai Jin, and Yu Zhang. 2021. HMvisor: Dynamic hybrid memory management for virtual machines. *Science China Information Sciences* 64, 9 (2021), 1–16.

[78] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*. 169–182.

[79] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Hai Jin, Xiaofei Liao, and Yan Solihin. 2022. Preserving Addressability Upon GC-Triggered Data Movements on Non-Volatile Memory. *ACM Transactions on Architecture and Code Optimization* 19, 2 (2022), 1–26.

[80] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, and Yan Solihin. 2021. Hardware-based address-centric acceleration of key-value store. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 736–748.

[81] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, and Yan Solihin. 2021. Supporting legacy libraries on non-volatile memory: a user-transparent approach. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture*. 443–455.

[82] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 261–272.

[83] Wen Zhang, Scott Shenker, and Irene Zhang. 2020. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 1029–1046.

[84] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*. 89–102.

[85] Zhan Zhang, Jianhui Yue, Xiaofei Liao, and Hai Jin. 2021. Efficient Hardware Redo Logging for Secure Persistent Memory. In *Proceedings of the IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application*. 41–48.

[86] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the International Conference on Management of Data*. 2653–2666.

Received 2022-07-07; accepted 2022-09-22