ENTS: Flush-and-Fence-Free Failure Atomic Transactions

Yun Joon Soh
UC San Diego
San Diego, USA
yjsoh@ucsd.edu

Steven Swanson
UC San Diego
San Diego, USA
sjswanson@ucsd.edu

Jishen Zhao
UC San Diego
San Diego, USA
jzhao@ucsd.edu

ABSTRACT

Persistent memory (PMEM) offers applications with DRAM-like performance and non-volatility. Yet programmers must ensure crash consistency — guaranteeing data consistency upon a sudden power failure — when implementing PMEM programs by carefully placing flush and fence instructions. That often introduces performance overhead and increases the risk of bugs. Previous approaches to reducing flush and fence instructions either require extensive hardware modifications or are only partially successful.

We propose Extricated Non-Temporal Store (ENTS), a programming library that removes explicit flush and fence instructions and provides an easy-to-use and low-overhead interface for failure atomic transactional PMEM programs without hardware modification. ENTS achieves crash consistency with a novel technique, Persist-On-Write (POW), which issues a fence-free non-temporal store (cache bypassing store) on each volatile data object upon modification. Instead of an explicit fence instruction, ENTS leverages fence-like instructions (e.g., `lock` or `xchg` under `x86_64`), which are already prevalent in the failure atomic transactional programs' concurrency control mechanisms. We evaluate ENTS on seven workloads: four data structures (B+Tree, RBTree, Hashmap, Skiplist), two transaction benchmarks (TPC-C, TATP), and a write-optimized hashtable (Level-Hash). Programs gain 1.8× and 2.1× higher throughput than Clobber-NVM (a compiler-directed crash-consistency tool) and PMDK (an industry-standard library), respectively.

CCS CONCEPTS

• **Information systems** → **Phase change memory**; • **Software and its engineering** → *Software fault tolerance*; Consistency.

KEYWORDS

Persistent Memory, Crash Consistency, Failure Atomic Transaction

ACM Reference Format:
Yun Joon Soh, Steven Swanson, and Jishen Zhao. 2023. ENTS: Flush-and-Fence-Free Failure Atomic Transactions. In *The International Symposium on Memory Systems (MEMSYS '23), October 02–05, 2023, Alexandria, VA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3631882.3631907

MEMSYS '23, October 02–05, 2023, Alexandria, VA, USA

ACM ISBN 979-8-4007-1644-7/23/10.
https://doi.org/10.1145/3631882.3631907

1 INTRODUCTION

Persistent memory (PMEM) is a byte-addressable, non-volatile memory device with comparable DRAM performance, filling the performance gap between fast, volatile DRAM and slow, non-volatile storage devices. To leverage the non-volatility of PMEM, a PMEM program must guarantee *crash consistency* — the data is recoverable to a consistent state upon a sudden power failure.

Without a proper crash consistency mechanism, a sudden power failure would leave irrecoverable inconsistent data. A typical method for ensuring consistency after a crash is to save information related to recovery before updating actual data. Prior works refer to such ordering requirements as *Persistent Memory Order (PMO)*.

Providing PMO incurs both the performance and programmability overhead due to computationally expensive flush and fence instructions. The former flushes dirty data from the CPU cache, and the latter stalls program progress until the PMO is guaranteed, wasting valuable CPU cycles. Misuse/overuse of these instructions may cause hard-to-detect bugs, burdening the programmer to correctly understand each instruction's semantics for the given hardware.

Prior works attempted to reduce the PMO overhead by either proposing hardware modification or designing a new programming model. Although it seems natural to replace the expensive hardware instruction at the hardware level, these works make intrusive hardware modifications, such as adding volatile/non-volatile buffers or extending existing coherency protocols [11, 15, 37, 40, 42]. Intel also acknowledged the program and proposed the extended Asynchronous Refresh Domain (eADR), which guarantees that globally visible stores — even dirty cachelines — are persistent [21]. Unlike the common belief, programmers still need fence instruction due to the cache bypassing store (Section 2.2.6). Software solutions alleviated the programming burden with various techniques, including speculation [26, 48] and compiler-level code injection [24, 33, 46]. However, these solutions still relied on fence instruction impacting performance and did not consider the eADR, resulting in complicated reasoning to optimize for the new hardware environment.

In an attempt to reduce PMO overhead for ADR/eADR, we made two observations: (1) non-temporal store (NTStore) performs well for both platforms and (2) programs frequently execute fence-like instructions, which are prevalent in concurrency controls and syscalls. Instead of going through the CPU cache, NTStore payloads are gathered in a *Line Fill Buffer (LFB)*, a hardware buffer dedicated to NTStore. For such reason, NTStore is often referred to as a cache-bypassing store and results in better performance than a temporal store in specific use cases (e.g., low temporal locality, large consecutive writes). The NTStore behaves the same for both ADR/eADR platforms and does not demand platform-specific optimization. Despite these benefits, NTStore is not heavily used in PMEM programming because it is difficult to reason about the correctness as

NTStore is weakly ordered; the NTStore may be executed in an unintended order.

Our second observation alleviates the programming burden when using NTStore; instead of asking the programmer to correctly insert fence instructions around NTStore, we observe that programs frequently execute serializing instructions. The serializing instructions, often used when implementing locks, syscalls, read-modify-write instructions, etc., drain internal CPU buffers related to memory writes, including the LFB. In other words, NTStore issued within a lock-delineated critical section is complete before exiting the section. Such observation fits naturally with a widely adopted PMEM programming model, lock-based *Failure Atomic SEction (FASE)* transactions, in which a programmer expects all the writes within the outermost locks to be atomically executed.

From the two observations, we propose ***Extricated Non-Temporal Store (ENTS)***, a performant flush-and-fence-free PMEM programming library for lock-based failure atomic transactions. ENTS provides a compatible interface to conventional PMEM persistence idioms for ease of use, does not invoke an explicit flush or fence within the library, and exhibits high performance on both ADR/eADR platforms without platform-specific optimization, all without hardware modification. The ENTS APIs include `pow_store` and `pow_epoch` to replace store-flush and store-flush-fence idioms, respectively. To exempt from explicit flush and fence instruction, ENTS relies on NTStore and implicitly executed serializing instructions. For high performance, ENTS segregates the DRAM/PMEM usage; the programmer-defined data structure resides on DRAM, and ENTS eagerly persists dirty data objects onto log-structured PMEM files using fence-free NTStore. The segregated design provides faster read (DRAM read latency < PMEM read latency), better NTStore usage (sequential NTStore on consecutive PMEM), and less cache pollution (NTStore bypasses cache). Instead of modifying the hardware, ENTS leverages an already implemented instruction: NTStore and serializing instructions.

We evaluated ENTS on data structures (B+Tree, RBTree, Hashmap, Skiplist), transaction benchmarks (TPC-C, TATP), and a widely evaluated hash table (level-hash [52]) on ADR and emulated eADR configuration. On emulated eADR, ENTS achieves 2.1× higher throughput than the industry standard programming library PMDK and 1.8× higher throughput than Clobber-NVM [46], a compiler-directed crash consistency tool. The compared works are properly modified for eADR by avoiding manual flush.

We make the following contributions:

- We observe that FASE programs execute various fence-like instructions, providing an opportunity to eliminate explicit fence instructions from a PMEM program.
- We provide ENTS, a novel flush-and-fence-free programming library with high performance regardless of the persistence domain (ADR/eADR).
- We retain the conventional programming idioms for programmability and assume a widely adopted lock-based PMEM programming model, FASE.
- We evaluate performance for various data structures and transactional benchmarks (lock-based data structures, TPC-C [45], TATP [43], and Level-Hash [52]) along with the recovery performance study.

2 BACKGROUND

2.1 PMEM Programming Model

2.1.1 FASE Models. Prior works identified two types of Failure Atomic SEction (FASE) programming models, where one assumes a strict isolation in which no thread dependency within a FASE is present, and the other relaxes the isolation requirement and permits more complicated concurrency controls, including cross-locking (hand-over-hand-locking). From the recovery's perspective, the former is easier to reason the correctness because each thread can independently determine the last consistent state. On the other hand, a recovery for the later model may involve rolling back a completed FASE if at least one of the dependent threads was incomplete before a failure. Various solutions (especially with the idea of committing a FASE as soon as it begins and recovering them with recovery-via-resumption) provided failure atomicity to programs written with the later model [24, 33]. In this work, we focus on the former model as with prior work [46], where we assume that the programmer provided sufficient thread isolation so that a committed transaction does not have to be rolled back.

2.1.2 Committing Failure Atomic Transaction. There are two ways to ensure a performant, correct transaction commits: (1) synchronously persist data before committing a transaction, and (2) speculatively persist data (as in prior works [26, 42]) and fix the wrong speculation later. The former simplifies the recovery algorithm at the cost of synchronizing the data movements. The latter improves performance with speculative stores but complicates the recovery process. ENTS takes the benefit of both by leveraging NTStore and fence-like instructions. ENTS relies on weakly ordered NTStore to asynchronously persist data for performance until a thread executes a serializing instruction. Specifically, for the assumed lock-based FASE model, a thread must execute the `unlock` before committing a transaction.

2.2 Architectural Background

2.2.1 CPU Cache Flush. Traditional non-volatile memories passed data through the OS page cache and thus persisted at page granularity with `msync()`. For byte-addressable PMEM, `msync()` incurs high overhead as the page must be flushed to the media as a whole, even with a single-byte update. To alleviate performance penalties, new instructions were introduced to flush data at finer granularity.

2.2.2 Memory Barrier. The most performant flush instruction, `clwb`, is asynchronous and thus requires memory barriers such as `sfence`, `lfence`, or `mfence`. These instructions wait until previously issued memory instructions retire, creating a global serialization point. If abused, it incurs a significant performance penalty, causing a performance bug. Hardware may reorder stores if not used where needed, and a sudden failure may result in an unexpected, irrecoverable PMEM state (correctness bug). In ENTS, misuse/overuse has a smaller performance impact as it does not manually flush dirty cachelines.

2.2.3 Synchronization Instructions. A fence instruction is not the only instruction that orders memory operations. Intel's Software Development Manual describes that the CPU avoids reordering reads and writes with I/O instructions, locked instructions, and

serializing instructions. As many programs use locks and atomic instructions for concurrency controls, the CPU persists fence-free NTStore before exiting a critical section. Even for a single-threaded program, a program invokes a syscall to commit a transaction (e.g., `write` to file, socket, device). Inside the syscall, kernel-level synchronization enforces hardware buffer drainage. Therefore, the CPU drains the fence-free NTStore before making the modification visible to other threads.

2.2.4 ADR/eADR. The asynchronous DRAM Refresh (ADR) and extended ADR (eADR) domains define when the data is guaranteed to become persistent. ADR domain guarantees that if the data reaches the Write Pending Queue (WPQ) in the integrated memory controller (iMC) within the CPU, it is considered persistent even though it has not yet reached the PMEM media. In the ADR domain, the cache hierarchy is still volatile, and it is up to the programmer to flush and persist the cached data. eADR extends the persistent domain to the CPU cache [21]. Upon a sudden power failure, hardware guarantees that the globally visible stores eventually reach the PMEM media before completely shutting down [21].

2.2.5 Temporal Store. Temporal stores are first stored in the cache (after shortly residing in the store buffer) and wait to be either evicted due to cache management or manually flushed with CPU instructions. Under the ADR domain, data is lost if it does not reach the Write Pending Queue (WPQ) before a sudden power failure; data in the cache and data flushed but in-flight to WPQ are all lost. Under the eADR domain, globally visible stores (i.e., part of the coherency domain) are flushed with flush-like instructions before the power failure [21]. Due to Total Store Ordering (TSO) under x86_64, temporal stores are globally visible in the same order as they are issued (stores are transitively visible).

A program that solely uses temporal stores could completely remove `sfence` under eADR, but it degrades the performance under eADR for two reasons: (1) it may fill up the cache with low temporal locality data, and (2) NTStore outperforms temporal store for large sequential writes [25, 49]. Many programs [2, 4, 12, 33, 35] use NTStore to persist logs onto PMEM, and by default, PMDK also uses NTStore when storing more than 256B (default `PMEM_MOVNT_THRESHOLD` value).

2.2.6 Non-temporal Store. Non-temporal store (NTStore) bypasses the CPU cache and moves the data directly to the destination. Before moving them to the PMEM media through iMC, they are gathered in a small Line Fill Buffer (LFB). Intel CPUs have several LFBs, each of 64 bytes. Upon an LFB eviction, it transfers the whole buffer in a single bus transaction if every chunk (8 byte) is valid. Otherwise, it performs multiple 8-byte memory bus transactions (called "partial write" [23]). Although LFB eviction is weakly ordered with respect to one another, eviction is atomic for the fully filled buffers. Unlike a temporal store, NTStore is still susceptible to reordering as Total Store Ordering (TSO) under x86 excludes it from the scope. Memory barriers should follow NTStore for global visibility, even under the eADR domain. For example, in PMDK, pmem_memcpy_persist always issues fence instructions regardless of ADR or eADR because it internally calls `memcpy()`, which leverages NTStore for large data.

2.3 Motivation

The motivation for our work stems from an observation that prevalent serializing instructions cause the NTStore (weakly-ordered, cache-bypassing store) to behave as a reasonably ordered store.

Optimizing a Program for eADR The advent of the extended Asynchronous DRAM Refresh (eADR) domain poses a new programming challenge: a performant programming model for both ADR and eADR. The temporal-store-only program is relatively easy to write for both ADR and eADR (remap the flush and fence to `noop` under eADR). However, it loses performance where NTStore performs better (e.g., large sequential write). A common misconception about eADR is that every fence could be safely removed. If the program directly or indirectly (e.g., external library) uses a non-temporal store, the programmer must inspect closely to decide on which fences to keep for eADR. With eADR around the corner, PMEM programming models should consider their performance under eADR-supporting hardware.

Natural Fit for FASE Transaction Our observation fits naturally with lock-based FASE transactions because the `unlock` operation is the last operation of FASE, and it internally executes serializing instructions. We grasped onto such similarity and created a performant failure atomic transaction library that exhibits high performance regardless of the underlying platform while preserving the conventional persistence idioms.

3 DESIGN

ENTS is a programming library that maintains the data structure in DRAM and appends dirty objects to the log-structured files on PMEM. By cleverly leveraging the non-temporal store (which does not need an explicit flush) and prevalent serializing instructions (fence-like hardware buffer drainage), ENTS can retain conventional persistence idioms without explicit flush and fence instructions, resulting in small performance overhead.

3.1 Design Overview and Challenges

The goal of ENTS is to provide low overhead crash consistency for the lock-based Failure-Atomic SEction (FASE) transaction model. The FASE model guarantees Atomicity, Consistency, Isolation, and Durability (ACID) per FASE. A programmer would enforce isolation with locks and expects ENTS to provide atomicity, consistency, and durability for updates within a FASE. As defined in prior works [3, 6, 24, 33], FASE is a region of code defined by the outermost locks. Throughout the paper, we would interchangeably use "FASE" and "transaction" to refer to the lock-based FASE transaction, and a transaction is committed when it exits the critical section.

Unlike prior works, which relied on time-consuming flush and fence instructions to ensure ACD property (without isolation, which the programmer provides) before a commit, ENTS uses fence-free NTStore to persist dirty data objects (treating them as log entries) onto a per data structure PMEM log file. During recovery, ENTS cleans any uncommitted log entries so that the recovered state is at the transaction boundary. Programmer-defined recovery threads may replay the logs using the globally consistent version information embedded in each log entry.

We explain ENTS at a high level using the example in Figure 1. (a) shows the initial state where there are three data structure

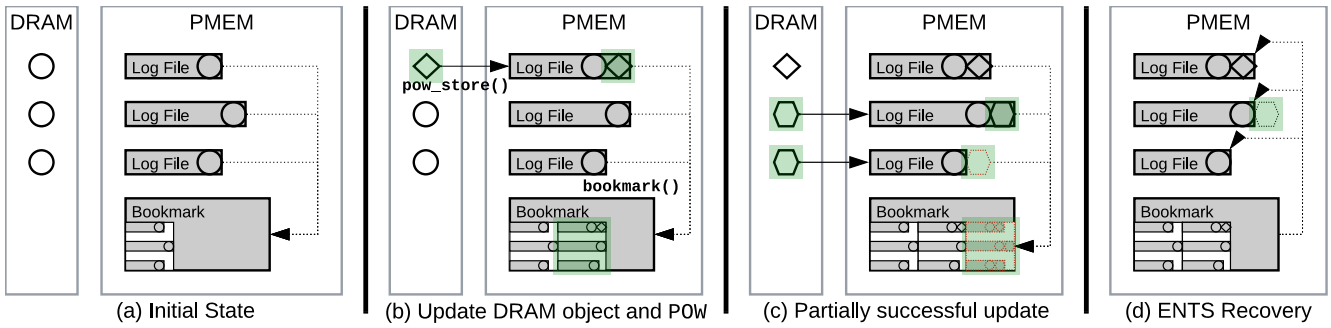

Figure 1: ENTS Design Overview.

objects whose values are a circle. Each data structure is associated with a log file, and a bookmark has a snapshot of the current state. In (b), one of the data objects was updated with `pow_epoch` (= `pow_store` + `bookmark()`), which would append the dirty object to the associated log file and update the bookmark object. In (c), while updating two objects, only one atomically persisted (dotted: partial success). However, ENTS can recover to the (d) state using the bookmark object.

Challenges ENTS's design challenges are (1) ensuring that the data structure can be log-structured for feasibility, (2) retaining conventional persistence idioms for programmability, (3) avoiding the reordering that may cause *false commit* (committing a transaction before data becomes durable) for correctness, and (4) designing a recovery algorithm that can identify invalid (incomplete or orphaned) log entries for crash consistency.

Feasibility To ensure that a data structure can be log-structured, ENTS enforces simple data layout rules. The rules allow ENTS to regard each data object as a versioned log entry. We discuss data layout rules and how ENTS leverages each rule in Section 3.2.

Programmability To comply with conventional persistence idioms, ENTS supports two types of flush-and-fence-free APIs (`pow_store` and `pow_epoch`), where each replaces the `store-flush` and `store-flush-fence`, respectively. Unlike the former, the latter transparently persists an additional consistency tracking metadata. We discuss the ENTS APIs, a high-level description of key functions, and their usability as idiom replacements in Section 3.3.

No False Commit To avoid false commit, ENTS leverages the observation that the last transaction operation, `unlock`, internally calls serializing instructions. The serializing instructions drain in-flight data in hardware buffers and update the memory [23]. We empirically concluded that out-of-order persistence is unlikely for NTStore from an "emulated power unplug" experiment on real hardware and discuss the result in Section 3.4.

Crash Consistency Without fence instruction, designing a recovery algorithm becomes challenging due to partial or reordered writes. To detect partial writes, ENTS paves the PMEM log files with *canary values* during file allocation, a predefined value that may not appear in normal program execution (Section 4.5.3). When the ENTS recovery thread reads a canary value, it discards the log entry containing the canary value. To detect reordered writes, ENTS tracks the tails of all log files. Upon calling a `pow_epoch`, a store-flush-fence equivalent, ENTS persists the tail information. The ENTS recovery algorithm discards any inconsistent log entries. For concurrency, each log entry has a version ID embedded. From the consistent log, a program can reconstruct the volatile data representation by chronologically replaying the log entries. We elaborate on how the proposed technique can detect/recover a partial state to a recovered state, thus complying with the FASE model semantics in Section 3.5.

3.2 Log-Structurability - Data Layout Rule

The key ENTS design component is persisting a dirty data object as a versioned log entry. ENTS enforces these rules that are not intrusive in most cases:

(1) Declare an 8-byte global variable called *version ID*.
(2) Each data structure can incorporate an additional 8-byte field reserved for ENTS.
(3) Each data object is uniquely identifiable across executions.
(4) No race condition for a data object.

ENTS uses the version ID to track the global persistence ordering in concurrent execution. Before persisting a data object, ENTS updates the reserved data field with the version ID. ENTS assumes uniquely identifiable data objects to reconstruct the volatile data structure from logs. In an actual implementation, several techniques can ensure such requirements, such as fat pointers, virtual memory addresses without randomization, or programmer-defined unique IDs. Without a race-free assumption, a data object may be persisted with an incorrect version ID. ENTS accesses and updates the version ID with read-modify-write instructions.

Programs that cannot enforce the data layout requirements are beyond the scope of the ENTS implementation in this paper. Some examples are a program that cannot share a global variable among its threads and a data structure with no room for additional data fields, possibly due to a highly optimized data layout. ENTS cannot deal with a data structure design where including a unique identifier for each data object is infeasible.

Another noteworthy assumption is that data structures are fixed-size. Although this seems restrictive, many applications rely on such an assumption. For example, PMEM allocators classify allocation requests based on the requested size and opportunistically return a slightly larger memory to reduce external fragmentation [39]. Furthermore, many programs limit the maximum input size to prevent

overflows. Programmers can use these predefined maximum sizes when designing a program with only the fixed-size data structure.

3.3 Compatible with Conventional Idioms

We designed ENTS to minimize the programmability efforts, and thus, complying with conventional PMEM programming idioms is important. Two widely used, conventional PMEM programming idioms are `store-flush` and `store-flush-fence`. To replace each idiom, we propose `pow_store` and `pow_epoch`, respectively. We discuss what ENTS API is, when to use them, and how they retain the conventional persistence idioms.

As listed in Table 1, API functions are categorized into three: allocate/store/load logs. A programmer must properly allocate the log files, replace conventional persistence idioms with store APIs, and use the load APIs to reconstruct the DRAM representation.

Allocation The log file allocation is the most unconventional part of ENTS API. A programmer would allocate the log file per data structure and must pass on *canary values*, a byte sequence that would never appear during normal execution. The canary bit-width is identical to the hardware-guaranteed atomic store unit, typically 8-byte in modern hardware. We discuss how a programmer can achieve this, along with a fallback plan in Section 4.5.3.

Persist-On-Write (POW) is an asynchronous persisting technique that eagerly appends the whole dirty data object (treating it as a log entry) to the persistent log file. Allocating memory for a log entry is as simple as incrementing the log count. Then, the ENTS library properly updates the object's version ID, including the sequence number (Section 4.1). Note that log entries from a single transaction can have multiple sequence numbers and even a gap among the numbers, as another concurrent thread may increment the global sequence number. This is not a problem for ENTS because the sequence number is merely used to enforce log replay order per thread and not for atomicity or consistency.

POW is similar to logging as it leaves a trail of modification. But unlike logging, there is no ordering constraint for POW; POW can be reordered and still be correct. POW is similar to copy-on-write (COW) because both techniques leave the original object untouched. Unlike COW, which must persist the copy before updating, POW updates the volatile version before persisting the data object. The closest mechanism is incremental fine-grained checkpointing [7], but POW eagerly persists data without waiting until a program reaches a checkpoint epoch.

`pow_epoch` is a store-flush-fence replacement. We referred to the term "epoch" because it provides similar semantics to buffered epoch persistency [10, 27–29, 41]. In buffered epoch persistency, a set of stores within an epoch is considered persistent, even though the actual data movement may be deferred for performance.

Without an explicit barrier, an actual data movement may be reordered with consecutive instructions (similar to how BEP defers data movement). However, this is not a problem because the FASE transaction does not expect to recover to the intra-FASE state. For the last `pow_epoch` before the FASE commit, the trailing `unlock` would have internally executed the serializing instruction.

`pow_epoch` is implemented as a `pow_store` wrapper without any memory barriers. Instead, it transparently persists the thread-local view of transactions, called *bookmark*, before returning. Bookmark, which plays a key role during recovery, tracks the tail of all log files. Providing bookmark consistency is not important because the FASE transaction provides enough isolation so that a committed transaction does not have to be rolled back; other threads can append a new log entry to one of the log files, making the thread local bookmark stale but the corresponding recovery thread disregards data objects appended by others during the recovery. Like any other data structure in ENTS, the bookmark object embeds version ID, and a recovery thread can chronologically replay log entries.

Load A programmer can use load APIs to reconstruct the volatile representation. ENTS guarantees that all log entries are valid and consistent up to a specific transaction boundary.

3.4 Ensuring Durable Data Before Commit

We empirically prove that NTStore reordering is rare with an "emulated power unplug" experiment. In this experiment, we emulate a power failure on a machine with threads sequentially writing to PMEM. We observed reordered writes for threads executing temporal store, whereas none for NTStore. We concluded that the small LFB, a small NTStore buffer on the datapath to PMEM, eagerly flushes data to PMEM. The flushing rate is eager enough that we do not observe reordered NTStore in our experiments.

3.4.1 Single-threaded Single-destination. To show empirical evidence that NTStore rarely persists out-of-order, we emulated power failure with `IPMITool` [31] while executing sequential NTStore at 8-byte granularity (guaranteed atomic store bit-width in the tested system). We varied the guaranteed minimum NTStore loop iterations before sending the power cycle signal. We leveraged our canary technique for out-of-order-persistence detection: `mmap()` a PMEM file and `memset()` the whole region with a specific canary value. When inspecting the file after the power cycle, we found an out-of-order persistence if the canary values surround a non-canary sequence. Among the 520 files, none of them persisted out-of-order.

3.4.2 Single-threaded Multi-destination - Small. We further tested with NTStore writing to interleaving destinations: 2, 4, and 8 different files. All 7280 ($(2+4+8)\times 520$) files persisted sequentially, even across the interleaved files. We know that the interleaved files did not persist out-of-order because the maximum difference of the last non-canary value offset was at most one. We denote an experiment as NTStore×*n*, where *n* denotes the interleaving factor. Listing 1 is an example of NTStore×2.

```
// Pass on different mmap'ed pmem files (pmemAddr array)
long long mm = 0;
for (size_t i = 0; baseAddr + i < fileEndAddr; i++) {
 _mm_stream_si64((long long *)targ->pmemAddr[0] + i, mm);
 _mm_stream_si64((long long *)targ->pmemAddr[1] + i, mm);
}
```

Listing 1: NTStore Experiment Sample Code

3.4.3 Single-threaded Multi-destination - Large. In NTStore×16 and NTStore×32 experiments, where the number of files is larger than the available LFB, we first observed the out-of-order persistence across destinations (i.e., a file may have more persistent data than others), but no reordering happened within each of the 6720 files. We believe partially filled LFBs are continuously preempted as more threads request LFBs than are available.

	Function	Semantics

Alloc	`POWFile *alloc_pf(char *name, size_t n, int64_t canary)`
	`POWFile *dealloc_pf(char *name, size_t n, int64_t canary)`
	`POWFile *realloc_pf(char *name, size_t n, int64_t canary)`
		Use the name as the key to update the log file descriptor map.

Store	`void pow_store(POWFile *pf, void *s, size_t n)`	Update the version ID and perform fence-free NTStore.
	`void pow_epoch(POWFile *pf, void *s, size_t n)`	Same as `pow_store` but additionally `bookmark()` before returning.
Load	`POWFile *load_pf(char *name)`	Return log file descriptor from the map using the name the key.
	`void *load_data(POWFile *pf, size_t index)`	Return `index`-th entry pointer.
	`void *load_latest_epoch;(POWFile *pf);`	Return pointer to the first epoch of the last transaction.
	`void *load_latest_data;(POWFile *pf);`	Return pointer to the last epoch of the last transaction.

Table 1: ENTS API Functions

3.4.4 Multi-threaded Single-destination. We emulated a multi-threaded ENTS scenario, where each thread performed NTStore×1 to its own destination. Among 280 files, we did not observe any reordering within a file. We did not search for reordering across files because distinguishing an out-of-order persist from a not-yet-executed is difficult, if not impossible; a thread may have been lucky to get scheduled in an ideal way to fully utilize the available LFBs.

We conclude that the NTStore is good at grabbing the LFB and requesting a premature (before it is fully combined) preemption may cause reordering. Furthermore, Intel SDM explicitly mentions that the hardware guarantees atomic eviction of write combining buffer [20]. From these facts, out-of-order persistence within a single file seems extremely rare for sequential NTStore, even without serializing instructions.

3.5 Crash Consistent Recovery

The recovery goal is to identify and erase inconsistent log entries, allowing the programmer to reconstruct consistent DRAM data from the consistent logs. We refer to the log entries in the recovered log files as *valid logs* and those discarded as *garbage logs*.

ENTS leverages the following insight during recovery. Because a log file is append-only, previous modification exists in chronological order. Therefore, if recovery to transaction t is possible for a given log file, recovery to an earlier transaction ($\leq t$) is also possible.

3.5.1 Recovery Algorithm. There are three recovery steps at a high level: cleanup of incomplete logs, finding a recoverable transaction boundary, and cleanup of logs generated after it. In the *tidy phase*, a recovery thread searches and cleans logs that are either partially persisted or persisted out-of-order. In the *agreement phase*, a recovery thread searches for the latest recoverable transaction commit. In the *truncate phase*, ENTS overwrites data log entries sequenced after the recoverable transaction. At this point, all log entries are intact (*tidy phase*), generated before the last transaction commit (*agreement phase*), and safe to replay because no incomplete logs are lingering (*truncate phase*).

3.5.2 Crash Consistency Correctness. We discuss how the described recovery provides transactional atomicity and consistency. We incrementally argue the correctness by elaborating how ENTS checks atomicity and consistency for a single log entry, multiple entries across files, and finally, multi-threaded executions.

Atomicity for a single log entry can be checked by (1) fixed-size log entry per data structure and (2) detecting partially persisted data with a persist-free consistency check (canary value, Section 4.5.3). Fixed-size log entry hints at the log boundary. A log entry has been persisted atomically if all of the 8-byte chunks of a log entry are not equal to the canary value. It is consistent because a thread modifies and persists an object as a log entry within a critical section.

For multiple POW to possibly different log files, `pow_epoch` records the tail of each log file in a special data structure called bookmark. A sequence of POW is atomically executed if all of the log entries up to the recorded tail are atomically persisted. The bookmark structure is allocated in thread-local storage, making it consistent with respect to other threads.

For multi-threaded scenarios, each recovery thread is associated with a thread ID generated before failure. Each thread disregards log entries if its version ID indicates a mismatch. By ignoring entries generated by other threads, a multi-threaded scenario is effectively identical to a single-threaded scenario.

3.5.3 Recovering Volatile Structure. We assume the volatile data structure can be reconstructed from a persisted data object array. Specifically, a non-singleton object must include a per-object ID. A programmer can recover the volatile structure by iterating the logs in increasing sequence order and copying each data object to the proper DRAM destination.

4 IMPLEMENTATION

ENTS is a crash consistent PMEM programming library that adds atomic durability to the lock-delineated failure atomic transaction for DRAM residing data structures. We elaborate on how ENTS leverages the described requirements in Section 3.2 to persist volatile data objects as log entries in append-only PMEM files. Then, we discuss ENTS API semantics and their usage. ENTS semantics builds upon extricated (fence-free) NTStore persistence and how ENTS leverages it correctly to provide correct crash consistency. Lastly, we describe the recovery implementation before providing the implementation details, such as the internal log file descriptor and canary value.

4.1 Volatile Data Object to Durable Log

ENTS treats each data object as a log entry and appends each entry to a designated PMEM log file for crash consistency.

Member	Description
`objname`	Unique name. A key for searching the file.
`objsize`	Size of each element (Unit: Bytes)
`canary`	An 8-byte predefined value. Never generated during normal execution.
`pow_idx`	POW file index within the descriptor map
`base_addr`	Pointer to start of the mmapped POW file.
`log_cnt`	Destination of the next POW call. Unit is the number of data objects. Only accessed with atomic Read-Modify-Write.

Table 2: Key Log File Descriptor Fields.

The key enabler is that every log entry, including the ENTS-managed metadata, has a *version ID*. A version ID comprises a canary protector, lock count, thread ID, and sequence number. **A canary protector** is a reserved bit that helps the programmer easily determine a canary value. It is defined per log file, and the value is fixed for all entries within the file. If the canary protector is set to 0, and the canary is $0xFF..FF$, the properly persisted version ID would not coincidentally match the canary value. **Lock count** is the number of locks held when calling POW. The **thread ID** is the thread ID of the current thread, which is cached upon thread spawn. A **sequence number** is globally visible and incremented with an atomic operation. Thus, all log entries across log files would have a unique sequence number.

We elaborate on how each rule in Section 3.2 contributes to the per-object log entry. With the first rule in Section 3.2, each data object hints at the global update order with its sequence number. The second rule provides enough per object memory for embedding the version ID. The third rule provides a means to recover the original data structure from an array of log entries. With the programmer-defined isolation (fourth rule), each thread updates the version ID and treats the object as a log entry.

4.2 Using ENTS API

4.2.1 ENTS API Implementation. We implement ENTS as a shared library, and it provides three categories of ENTS APIs: allocation, storing dirty data objects, and loading log entries for reconstruction (Table 1).

Log File Allocator The allocator functions take three parameters passed on by the programmer and return a log file descriptor (Table 2), which includes canary value, current `log_cnt`, and data structure size. The allocation process starts with checking the name conflict against existing descriptors. Internally, ENTS maintains a mapping between the `objname` and the corresponding descriptor. Then, it either inserts an entry to the map or updates an entry depending on the exact API function.

Once the volatile map has been updated, ENTS creates and maps a PMEM file where the `mmap()`'d address is recorded in a log file descriptor. Using the recorded address, the whole file is initialized with the canary value with an asynchronous NTStore. This process is highly concurrent, and we use eight threads by default. Lastly, it POW the updated log file descriptor to the metadata log file to make the allocation persistent. At this point, the descriptor can be passed on to store/load APIs for writing/reading a log entry.

POW `pow_store` and `pow_epoch` are wrappers for Intel's NTStore intrinsic functions. They take the log file descriptor and a pointer to the data object, along with its size. They first update the version ID of a data object by dereferencing the `ptr` parameter. After atomically incrementing the `log_cnt` with fetch-and-add ([44]), which naturally reserves an exclusive memory region, ENTS computes the destination pointer with the `base_addr`, `objsize`, and the fetched `log_cnt`. For `pow_epoch`, it additionally calls `bookmark()`, which atomically increments the global sequence number and persists the bookmark object to the bookmark log file.

Fence-free NTStore does not pollute the CPU cache and is faster than a temporal store for sequential writes, but it is weakly ordered, and aligning NTStore at the cache line has a meaningful performance impact [22]. If the data are not properly aligned, write-combing stores may issue multiple memory bus requests, degrading the performance. ENTS executes piece-wise sequential NTStore at a data object granularity. The object size is a multiple of LFB with programmer-inserted padding) to avoid amplified bus requests.

Load Load functions are categorized into two: loading log files and loading log entries. The former searches the log file descriptor map for the targetted `objname`. The latter category computes the VMA of data entry with the following formula: $base + index \times objsize$. We further provide helper functions that return the first and last data object of the last epoch.

4.2.2 Log Compaction/Garbage Collection. The programmer is responsible for online garbage collection and log compaction. When the program exhausts a POW file, it must reallocate the POW file and take it as an opportunity for log compaction. A typical program would halt upon capacity exhaustion, reallocate the POW file, iterate the data objects, and POW them to the new file.

For a program that cannot afford the "stop-the-world" garbage collection, the programmer can create background threads that maintain the "live" objects and opportunistically persist them to the compacted POW file. The program may maintain a "live" object table where each entry consists of the virtual memory address (VMA) of the object on DRAM, the VMA of `mmap()`'d PMEM log file, the size of the object, and a dirty bit. Whenever the program calls POW, it adds an entry to the list with a dirty bit set. The background thread monitors the dirty bit and opportunistically persists them to the garbage-free version. This implementation of garbage collection is left for future work.

4.2.3 Concrete Example. We provide a concrete example with a linked list. In the example, ENTS guarantees failure-atomicity upon epoch commit. Although ENTS does not include an explicit hardware memory barrier, `unlock` internally relies on memory synchronization for correctness [19]. Therefore, any NTStore within the critical section is persistent before exiting.

```
POWFile *listPF, *nodePF;
VID_T vid; // Version ID, rule 1

typedef struct LinkedList {
  VID_T vid; // rule 2
  Node* head;
  size_t len;
} LL; // rule 3 not needed for singleton
```

```
9
10 typedef struct Node {
11   VID_T vid; // rule 2
12   Node* curr; // rule 3
13   Node* next;
14   uint64_t value;
15 } Node;
16
17 void init() {
18   listPF = allocPF(..);
19   nodePF = allocPF(..);
20   /* Other initializations ... */
21 }
22
23 void insert(Node *prev, Node *next, Node *now) {
24   pthread_rwlock_wrlock(&lock, ..);
25   prev->next = now;
26   now->next = next;
27   pow_store(nodePF, prev, sizeof(Node));
28   pow_epoch(nodePF, now, sizeof(Node));
29   pthread_rwlock_unlock(&lock, ..);
30 }
31
32 void update(Node *a, int newVal) {
33   pthread_rwlock_wrlock(&lock, ..);
34   a->value = newVal;
35   pow_epoch(nodePF, a, sizeof(Node));
36   pthread_rwlock_unlock(&lock, ..);
37 }
```

Listing 2: LinkedList Example (Insert/Update)

Listing 2 shows a linked list example using ENTS. Lines 1 and 2 declare a global pointer to the log file and global version ID. Lines 5 and 11 reserve 8-byte for each data object. Line 12 stores a unique ID for each data object used for reconstructing volatile data structure. Unlike `Node`, `LinkedList` does not need a unique pointer value because it is a singleton. Lines 18-19 allocate log files. Lines 27-28 and 35 are where the POW occurs.

```
1  void recover(){
2    listPF = loadPF(..); int lenL = sizeof(LL);
3    nodePF = loadPF(..); int lenN = sizeof(Node);
4
5    Map<uint64_t, uint64_t> map; // remapper
6    LL *list = malloc(lenL); // LinkedList
7    memcpy(list, load_latest_data(listPF), lenL);
8
9    // Recover Nodes
10   Node *nodeL = malloc(lenN * list->len);
11   Node *ptr = nodeL;
12   for(int i = 0; i < nodePF->log_cnt; i++, ptr++){
13     Node *node_p = load_data(nodePF, i);
14     if(map[node_p->curr] == 0){
15       memcpy(ptr, node_p, lenN);
16       map[node_p->curr] = (uint64_t) ptr;
17     }
18     else{memcpy(map[node_p->curr], node_p, lenN);}
19   }
20
21   // Remap pointers
22   for(int i = 0; i < list->len; i++){
23     Node *n = nodeL[i];
24     n->curr = n;
25     n->next = (Node*) map[n->next];
26   }
27
28   // Off-line garbage collection/log compaction
29   POWFile *pf2 = realloc_powfile(..);
30   for(int i = 0; i < list->len - 1; i++){
31     Node *n = nodeL[i];
32     pow_store(pf2, n, lenN);
33   }
34   pow_epoch(pf2, nodeL[list->len - 1], lenN);
35 }
```

Listing 3: LinkedList Example (Recover)

Listing 3 is an example recovery code using the same data structure as Listing 2. In the example, the programmer leverages that the allocated node object is not reallocated to a new VMA during a single program execution. After recovering the latest `LinkedList` object from its log file in lines 6-7, the programmer allocates a large memory on DRAM (line 10). As the program iterates the node log file in lines 12-19, it loads data from the log file onto DRAM. Whenever it encounters a log entry for the same unique ID, it overwrites the whole data object, as in line 18. This is correct because ENTS persists in an append-only fashion, and thus, log entries closer to the end of the file reflect more recent changes. After loading the latest data, the program must update the reference between data objects. For example, a virtual address 0x100 may point to the head node in the previous program execution, but it may point to garbage in the current run. In the code, the mapping (line 5) constructed during the node recovery process (lines 12-19) is the hint to update the pointers correctly (lines 22-26). The last step is POW'ing the updated data objects. We elaborate on the garbage collection/log compaction opportunity in Section 4.2.2.

4.3 Atomic Durability — Extricated NTStore

ENTS heavily rely on NTStore without constraining them with manual fence instructions, and thus, the name extricated (from manual fences) NTStore. Extricated NTStore is not a silver bullet due to potential performance and correctness hazards. We describe the NTStore, why it may perform badly or even incorrectly, and our approach to avoid such scenarios.

NTStore bypasses the cache and instead gathers the payload in *Line-Fill Buffer (LFB)*. The exact number and size of the buffer are not architecturally defined, but modern CPUs have around 8-10 LFBs each of 64 bytes [23]. NTStore is weakly ordered because the buffer eviction may be deferred or eagerly triggered depending on the pressure on LFB or memory misalignment.

A rule of thumb for performant NTStore is to fully fill the LFB, at which the hardware will evict the buffer. A partially filled LFB would issue many more bus operations, whereas a full LFB may be evicted with a single bus operation. For ENTS, we align data to LFB size and pad the data objects to a multiple of LFB. We discuss the performance impact of padding in Section 5.4.

Using fence-free NTStore to overwrite a memory region may cause unexpected behavior impacting the correctness. NTStore is weakly ordered, meaning that an NTStore issued earlier (older instruction) may be completed after the NTStore issued later (younger instruction). ENTS avoid such scenarios with the append-only log-structured design; persist order does not impact correctness as long as they are all persistent before exiting the critical section.

4.4 Recovery Implementation

ENTS recovery works in three phases: cleanup of partially persisted log entries, finding a recovery point, and truncating log entries sequenced after the recovery point. ENTS spawns a recovery thread whenever it identifies a newly discovered thread ID for metadata or bookmark log entry. The spawned thread ignores log entries that are not associated with its targeting thread ID. Such a design creates a hallucination that log files are created per thread even when the log entries with different thread IDs are mixed within the same file.

During the *tidy phase*, a recovery thread iterates each log file (which may be interpreted as per thread file by ignoring entries irrelevant to the thread) and finds a *hole* VMA (the address of the first properly aligned canary value). For subsequent entries, a recovery thread overwrites log entries that match its thread ID and are sequenced after the *hole* with the canary value. In the *agreement phase*, the recovery procedure iterates the bookmark log file in reverse order until it finds the first recoverable epoch. An epoch is recoverable if the `log_cnt` of each log file is no smaller than the value in the bookmark and the lock count is zero. The sequence number of the recoverable epoch is the *consensus eid*. During the *truncate phase*, it overwrites any data with a sequence number larger than the *consensus eid* with the canary value. The recovery procedure is resilient to failure because cleaning the partially persisted logs or truncating the inconsistent data does not affect the *agreement phase*.

4.5 Library Implementation

4.5.1 Library Initialization. A programmer would link to the ENTS library dynamically when building the binary. Most ENTS logic is executed upon a library load, including the recovery process. Before initiating the recovery process, the ENTS library traverses the file hierarchy, starting from the metadata log file information, then the bookmark file, and finally, the associated log files. We detail the hierarchy traversal.

Metadata The first step during the library initialization is finding the metadata log file containing information about all allocated bookmark data and log files. ENTS uses the environment path variable that points to the root ENTS folder, and the entry log file is called a metadata log file. Once located, it generates a map where a key is the name of a data structure, and the value is a log file descriptor. The metadata file is also persisted with POW, and ENTS reconstructs the map by copying each log entry into DRAM, treating the entry's object name as the key and the object itself as a value. We refer to the reconstructed map as a **log file descriptor map**, which is internally represented as an array of log file descriptors.

Bookmark File The next step is identifying the bookmark file. Within the descriptor map, ENTS searches for a special descriptor with a key value of ".bookmark." This log file holds `struct Bookmark` objects. Like any other log entry, entries have an embedded version ID along with an array of `log_cnt` where the i-th element indicates the thread-local `log_cnt` of the i-th POW File at the moment of `pow_epoch`. A thread does not have to POW the bookmark with the latest log count information because the goal is to find the **least** log count needed to recover to a certain version. Another thread may have incremented the log count but does not impact the per-thread recovery.

Per-thread Bookmark While iterating the bookmark file, the library initializer spawns a thread whenever it reads a bookmark object from a previously untracked thread ID. The spawned recovery thread is associated with the triggering thread ID and is responsible for recovering all log entries with the same thread ID.

4.5.2 Log File Descriptor. We provide some noteworthy member fields of `struct POWFile` in Table 2. This object is relevant to ENTS bookkeeping, and the program should not directly update it. Read-only variables (`objname`, `objsize`, `canary`, `pow_idx`) are initialized and fixed upon POW file allocation. `base_addr` is the `mmap()`'d address and is never overwritten during the program runtime. The `log_cnt` indicates the total number of data entries within the file. Internally, `((char*)base_addr) + objsize` $\times$ `log_cnt` gives the VMA for subsequent POW.

4.5.3 Canary Value. The programmer defines a per-data-structure canary value: an 8-byte aligned value (because PMEM supports 8-byte atomic read/write) that a program would never generate during the normal execution. A canary value in a POW file indicates either a partial persistence or an end of data.

For a data field smaller than eight bytes, a programmer can round each field to 8 bytes with paddings. Once a programmer confirms that padding bits are consistently set to zero, they can consider an 8-byte value with all bits set to one as a canary value.

Fallback Plan When intentional misalignment is insufficient, a programmer can prepend padding to a data field and align the padded data to 8 bytes. Due to the padding, a program can avoid coincidentally generating an 8-byte aligned canary value. We included the sub-optimal scenario in the level-hash evaluation (Section 5.5).

4.5.4 Thread-Safety and Concurrency. ENTS is thread-safe because it modifies shared variables (e.g., `log_cnt`, global sequence number) with atomic read-modify-write (RMW). If a program prevents multiple threads from concurrently modifying the same log file, a programmer can turn off the concurrency support for better performance. For example, in Listing 2, the `nodePF` is only modified when a thread acquires a writer lock. Turning off RMW would remove unnecessary RMW costs.

Thread-local Bookmark Object We clarify one potential concern regarding the concurrent access to the bookmark. Each thread has its own thread-local bookmark object but persists as a log entry onto the same Bookmark file. This may result in a non-monotonically increasing sequence number of bookmark logs. However, during recovery, each thread concerns log entries associated with a specific thread ID. Therefore, it does not cause a correctness issue since a single thread (1) exhibits sequential consistency at FASE granularity and (2) does not rely on a continuous sequence number for correctness — but rather on the existence of all log entries sequenced before the consensus version ID.

Lock Count ENTS wraps the pthread `lock`/`unlock` calls to increment/decrement the thread-local lock count after/before the real execution, respectively. Inside the unlock, ENTS checks if the current lock count is zero and bookmarks with the updated lock count. This process is transparent to the programmer.

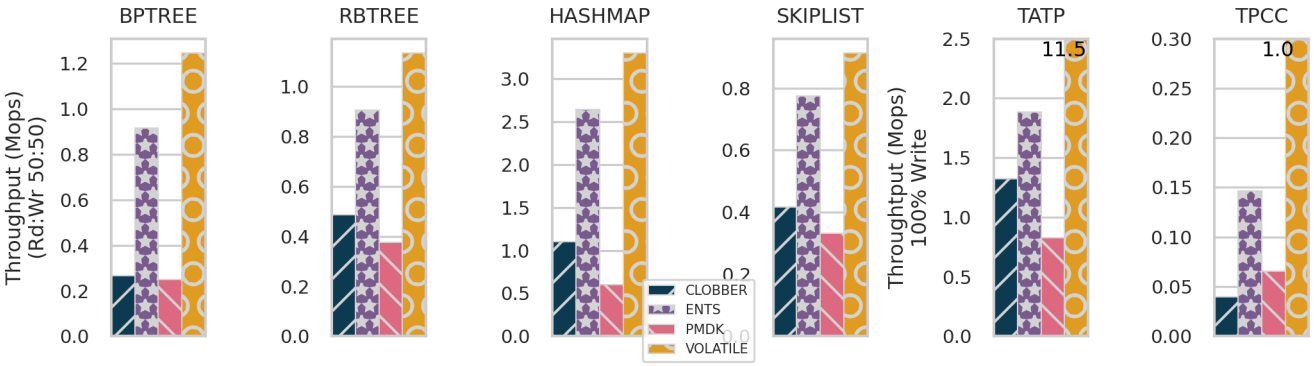

Figure 2: Single Threaded Performance on ADR

Data Structure	Concurrency	Source
`B+Tree`	Reader-Writer Lock Per Node	Clobber-NVM
`RBTree`	Global Reader-Writer Lock	Clobber-NVM
`HashMap`	Reader-Writer Lock Per Bucket	Clobber-NVM
`Skiplist`	Reader-Writer Lock Per List	Clobber-NVM
`TPC-C`	Per Transaction Lock	Janus
`TATP`	Per Transaction Lock	Janus
`Level-Hash`	Single Threaded	Level-Hash

Table 3: Workload Detail

5 EVALUATION

We evaluated the ENTS against Intel's PMDK [9] and software-only PMEM programming model Clobber-NVM [46] on ADR and emulated eADR applied to the workloads in Table 3.

The goal of the evaluation is to answer the following questions.

- How does ENTS perform against other crash consistency solutions?
- Does ENTS performance scale?
- Which design factors of ENTS contribute to the performance?
- How would ENTS recovery perform concerning the log file size?

5.1 Experiment Setup

We ran our experiments on a single-socket machine powered by a 20-core Cascade Lake Intel Xeon Gold 6230 CPU. For memory, we used the 16 GB DRAM and 128 GB PMEM per DIMM pair in a 2:2:2 configuration. We averaged over ten runs before computing the relative throughput. For eADR emulation, we used PMDK's `PMEM_NO_FLUSH` option.

5.2 Performance

5.2.1 ADR. To show how ENTS performs against other crash consistency techniques, we evaluated the single-thread 50:50 read/write performance (Figure 2) against PMDK [9] and Clobber-NVM [46]. ENTS outperforms both PMDK and Clobber-NVM by 2.3× and 1.9×, respectively. As we later show in Section 5.4, several factors contribute to the ENTS's performance. The idea of log structuring the data objects and eagerly triggering the fence-free NTStore, having a DRAM copy for faster memory access, and eliminating the manual fence instruction all contribute to the ENTS's high performance.

To measure the crash consistency overhead of each technique, we also compared against the volatile implementation of each benchmark. ENTS showed about a 35% slowdown, whereas PMDK and Clobber-NVM showed a 70% and 65% slowdown. This indicates that crash consistency is still a noticeable overhead even when most memory accesses are from DRAM and requires future research to further reduce the overhead.

All three techniques (PMDK, Clobber-NVM, and ENTS) showed a significant slowdown for transactional benchmarks (TATP and TPCC in Figure 2) for two reasons. First, these benchmarks are 100% write, whereas the data structure workloads (B+Tree, RBTree, Hashmap, Skiplist) were 50:50 read/write. Write-only would incur higher write latency on PMEM than on DRAM. Furthermore, it deprives the opportunity to persist data concurrently, which is a critical performance optimization for most PMEM programs.

5.2.2 Extended ADR. To understand how the eADR platform reduces the persistence overhead, we measured the performance gain by eliminating the manual flushes from the program. In short, PMDK showed a 21% performance gain on average, whereas only 4% for Clobber-NVM. This is because PMDK frequently flushes and fences write-ahead-logging, whereas Clobber-NVM reduces the number of logs with compiler analysis and logs only the data that may be overwritten. As expected, ENTS and Volatile do not involve manual flushes and thus did not benefit from the eADR. We only show the results for single-thread execution, but multi-threaded shows similar results.

5.3 Scalability

To understand how ENTS scales when compared to other solutions, we measured the aggregated throughput as we increased the number of threads. ENTS generally outperformed PMDK and Clobber-NVM by 2.1× and 1.8×, respectively, across the measured threads and workloads. For TATP, ENTS performed worse than the other techniques because TATP is a short transactional benchmark with a short time window to persist while executing the main program logic concurrently.

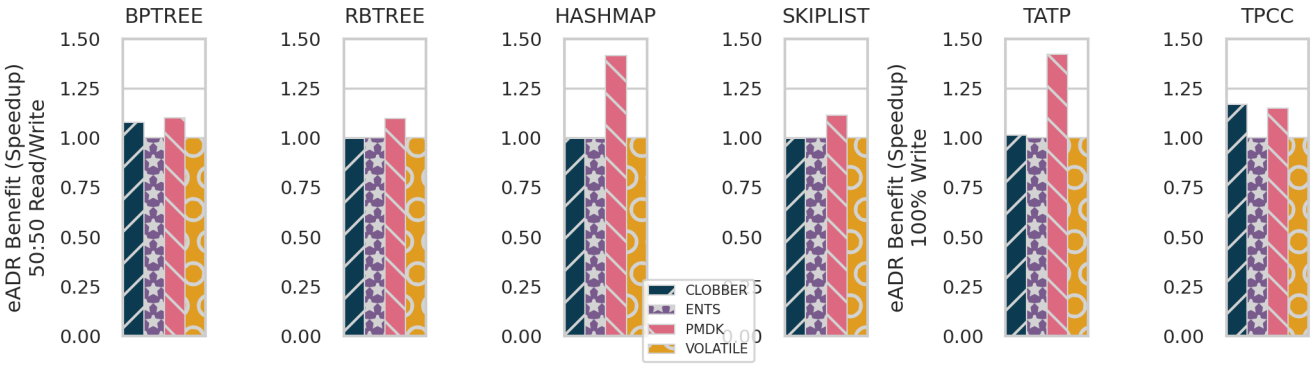

Figure 3: Single Threaded Performance Gain on eADR

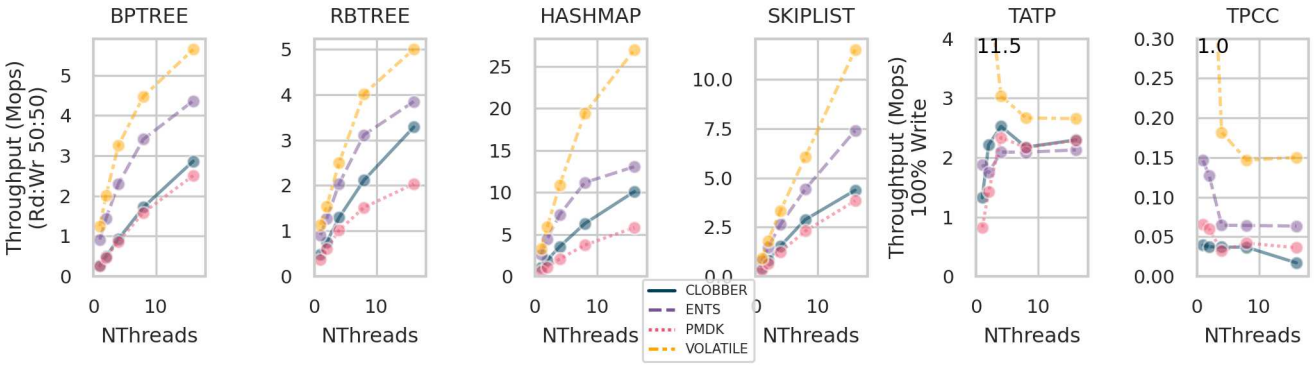

Figure 4: Scalability on ADR

Another interesting observation is that ENTS shows a stiff performance degradation when increasing the thread beyond eight threads. When the number of threads is larger than the number of available LFBs, a frequent partially filled LFB eviction may occur. As discussed in previous sections, partially filled LFB degrades the NTStore performance with multiple bus operations. We conclude that the scalability of the ENTS is bounded by the number of LFBs under currently available hardware.

5.4 ENTS Variants

To understand how ENTS design components impact its huge performance, we measured the performance of variant programs. To understand the impact of placing the main data on DRAM, we compare against a variant called *ENTS-PMEM*, which allocates the main data on PMEM as with Clobber-NVM and PMDK. To understand the impact of fence-free NTStore, we replaced the NTStore with Temporal-Store-Flush-Fence (TSFF), which we refer to as *ENTS-TSFF*. To understand the impact of both, we moved data onto PMEM using TSFF and denoted *ENTS-PMEM-TSFF*. Lastly, to understand how intentionally padding the data structures to catchline size can impact the performance, we measured the padded version for each variant.

Figure 5 shows the performance of ENTS variants. In general, the non-temporal version running on DRAM shows the best performance. One exception was the TPCC transaction workload, where the workload benefitted from the temporal locality. Writes in the TPCC workload were back-to-back to the same cacheline, and allocating the cache to gather updates before flushing the fencing was a better strategy than the default ENTS. Although default ENTS showed better performance, TATP workload performance also hints that it can benefit from write allocation. When putting the main data on PMEM (`ENTS-PMEM` and `ENTS-PMEM-TSFF`), the temporal store performed better than the non-temporal store strategy. For data structures, temporal locality did not benefit much because they typically traversed to another data object rather than repeatedly updating the same one. Regardless of whether the main data was on DRAM or PMEM or whether the temporal or non-temporal store was used, ENTS variants outperformed prior works.

Figure 6 shows the impact of padding each data structure. TATP and TPCC benefited the most from padding the data structures. This is because these workloads are write-only workloads with a small time window for concurrently evicting the LFB. In such a scenario, partially filled LFB may be evicted, incurring higher performance overhead. Padding the data structure fills the LFB fully, resulting in a smaller overhead.

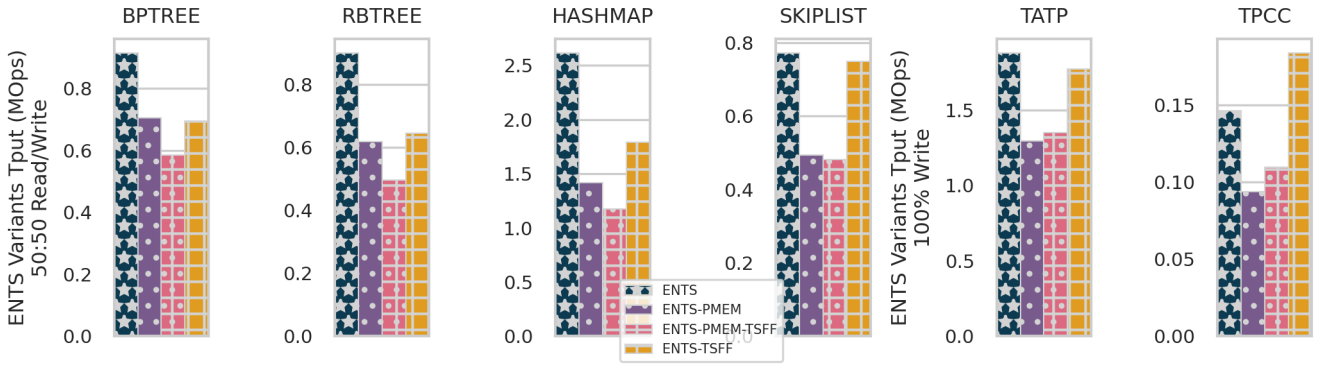

Figure 5: Performance of Unpadded ENTS Variants. Note that ENTS is the same ENTS from Figure 2.

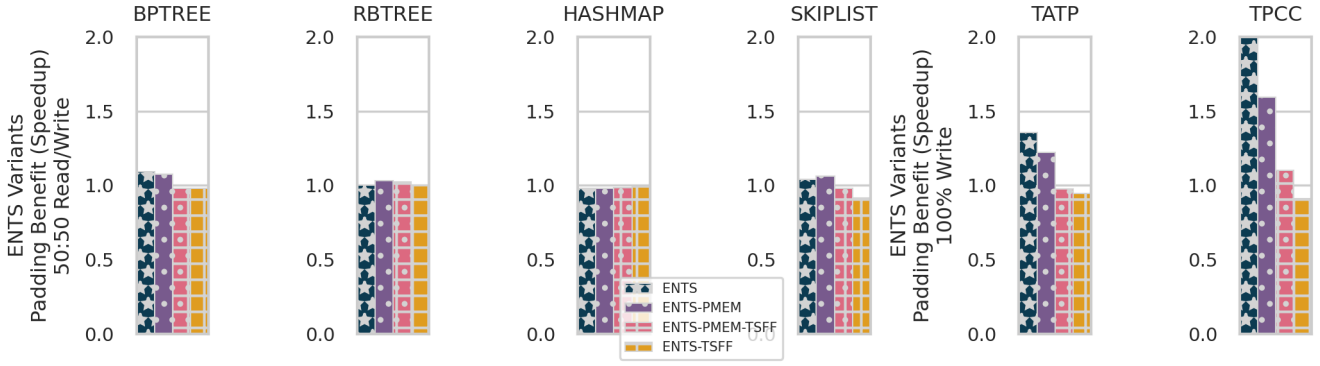

Figure 6: Speedup by Padding the ENTS Variants. Padded data structures at cacheline granularity.

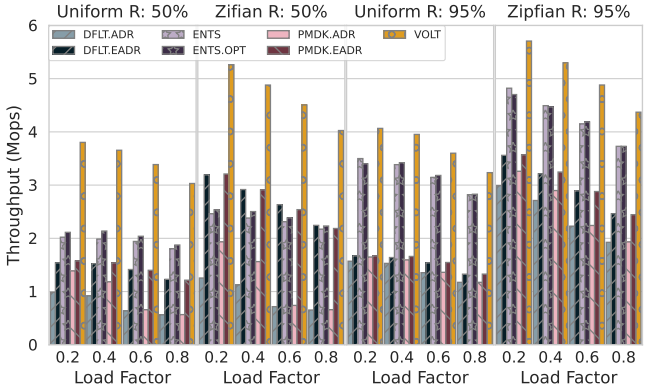

Figure 7: Level-Hash Performance using YCSB

As expected from the NTStore behavior, padding improved the ENTS and ENTS-PMEM performance by 20% and 14%, respectively. For temporal store-based implementations, the additional padding bytes became a 5% overhead for ENTS-TSFF. For ENTS-PMEM-TSFF, padding had negligible impact, resulting in a geometric mean of 0% performance gain.

5.5 Level-Hashing

To show the benefit of ENTS under a write-optimized single-threaded hash table, we applied ENTS to the level-hashing [52]. Level-hashing uses two-level structured KV buckets to defer the hash expansion as much as possible by moving around the entries using two hash functions [52]. Upon hash collision, it attempts to relocate the existing entry to another location within the same level using a secondary hash function. If insufficient, it makes a final attempt to move entries across the level and avoids the expensive hash expansion. We used the level-hash default configuration 31 B KV, 10 million entries initial capacity, with a minimum of 12 M operations per experiment, four associativities per bucket.

ENTS vs. ENTS.OPT We observed that ENTS.OPT, where each data structure is padded to cacheline size, performs better than ENTS for the write-heavy workload (left two subplots in Figure 7) but shows the reversed performance for the read-heavy workload (right two subplots in Figure 7). This is because ENTS must write a whole bucket per insertion, resulting in intense data movement. For a read-heavy workload, ENTS has a slight advantage over ENTS.OPT for small load factor where hash collision is less likely to occur. This gap wears out as the load factor increases, incurring frequent hash collision, thus, the data movement.

	nfile	objsize		holeat	filesize
		Bytes	log_2		
Tidy	0.351	0.027	0.130	-0.006	0.835
Agree	0.091	0.415	0.290	0.087	0.178
Trunc	0.903	0.005	0.098	0.172	0.209

Table 4: Correlation Coefficient Between Recovery Latency and Various Variables

ENTS vs. others We make the following observations about ENTS performance with respect to other implementations. ENTS performs worse than others for Zipfian (R: 50%) workload running on eADR. Zipfian workloads have high locality and PMDK.EADR and DFLT.EADR takes advantage of write coalescing at the CPU cache without touching the PMEM. The relatively larger gap between ADR and eADR for Zipfian (R: 50%) also supports this idea; eagerly flushing the modified cache line to PMEM (DFLT.ADR and PMDK.ADR) removes the opportunity to gather the updates in the CPU cache. ENTS shows a higher speedup for read-heavy workloads over write-heavy compared to DFLT and PMDK. This is because ENTS keeps a volatile copy on the DRAM for faster read access, whereas others read from the slower PMEM. Overall, ENTS outperforms the default (DFLT) by 2.3× and 1.4× under EADR and ADR, respectively.

5.6 Recovery Evaluation

To show ENTS recovery scales, we timed each recovery phase while varying the following variables: nfile, filesize, objsize, and holeat. We first initialized nfile POW Files each of filesize that holds data objects where its size is objsize. We then performed POW() on every file (pow_epoch on the last file) until they were half full. After manually inserting a canary value in one of the POW files so that data located at holeat gets cleaned up during the *tidy phase*. We measured time per GB for each phase separately.

Results show that the agreement and truncate phases are negligible (< 10 μs/GB) when compared to the tidy phases (from 113 to 281 ms/GB). During the library initialization, it would further increase the tidy phase latency as it checks every 8B chunk against the canary value. Although the actual recovery would involve additional reconstruction from the logs, recovery seldom occurs, and iterating the logs during recovery is common in other proposed solutions [14, 46].

Lastly, we show the correlation coefficient between the time it takes for each phase and the variables we tested (Table 4). Results show that the most time-consuming phase, the tidy phase, had a high correlation with the filesize as it reads every byte in every file.

6 RELATED WORK

We discuss existing works and how naively applying our observation — relying on serializing instructions in lieu of fence instruction — is incorrect for these works.

6.1 Crash Consistency Techniques

Previous works [17, 25, 36, 49] show that flushes and fences are expensive and degrade the overall performance. To minimize the cost, previous works entail techniques such as batching flushes [10, 17], performing hardware logging [16, 51], minimizing ordering requirements [17, 38], or performing lazy flushes [32, 34]. However, these works target either one of the persistence domain (ADR/eADR), modifies the underlying hardware, or can not leverage our observation.

Some works propose a separate data path, including a dedicated buffer for PMEM writes [1, 15, 26, 29, 37, 48], and our observation cannot be applied to these works to eliminate stalling. Even when the serializing instruction drains the proposed buffer, the intra-thread PMO requirement may stall the volatile execution at the epoch boundary. For example, Strandweaver [15] extended the existing ISA for ARMv7, which has a more relaxed consistency model than Total Store Order (TSO). Under strand persistency, a programmer defines a fine-grained ordering requirement (strand) using the proposed instructions and invokes a *persist barrier* to guarantee the epoch persistency within the strand. This *persist barrier* order writes within a strand and causes stalling.

Any logging models rely on the ordering between persisting log and in-place updates either explicitly through a fence-like instruction or implicitly handled by the hardware, causing execution stalls. For example, Synchronization Free Region (SFR) [14] proposed *SFR-atomicity* that batches the persist operation at the end of a synchronization region (similar to how epoch persistency provides epoch-atomicity). Decoupled-SFR, a relaxed version, performs asynchronous in-place updates when the undo-log is persistent. Instead of synchronizing the in-place update, the *pruner thread* running in the background periodically commits and prunes the logs. Decoupled-SFR is similar to ENTS in decoupling the VMO and PMO. However, decoupled-SFR still relies on fences for intra-thread PMO; the foreground thread persists (with flush-and-fence) the undo-log before updating in-place, and the *pruner thread* may stall to apply the logs in which they were generated. Unlike decoupled-SFR, the log-free ENTS does not rely on strict intra-thread PMO.

6.2 Persistency Model

Strict Persistency (SP) couples the persistent memory ordering (PMO) with volatile memory ordering (VMO). It removes the burden to handle the discrepancy between the two orderings, but the lack of concurrency hurts performance.

Buffered Strict Persistency (BSP) provides the same guarantee as SP by allowing persist buffers to accumulate the PMEM writes so that volatile execution proceeds without halts [1, 27, 29]. Data in these buffers are concurrently flushed to PMEM, retaining the VMO. A more recent work speculatively updates (i.e., PMO ≠ VMO) the PMEM and recovers from misspeculation with a failure-atomic transaction [26] or rewrites a cacheline with the saved data in the memory controller [48]. Although BSP can move some of the persist operations out of the critical path, it requires detailed reasoning about inevitable stalls to provide ***Strong Persist Atomicity (SPA)***: writes to the overlapping PMEM locations assumes VMO [40].

Epoch Persistency (EP) permits the reordering within but not across programmer-defined epochs [8]. Programmers define the epoch boundary using fence instruction where the hardware thread stalls and drains pending PMEM writes.

Buffered Epoch Persistency (BEP) gathers the writes in a dedicated buffer so the volatile execution proceeds without stalls at an intra-thread epoch boundary [10, 27–29, 41]. However, BEP must stall to provide SPA, often involving intrusive hardware modification to detect it.

Under **strand persistency**, programmers can articulate the PMO within a single thread by defining a logically independent instruction sequence called a *strand* [15, 40]. A single thread may have multiple *strands*, and each *strand* may concurrently persist while retaining the programmer-defined intra-strand ordering. As other BEP works, it stalls to drain buffer(s) and provides SPA.

Various works pointed out that inevitable stalls may occur to provide SPA [15, 27, 40]. Our design is free from SPA as it only appends to PMEM files.

6.3 Complicated PMEM Performance

PMEM is a byte-addressable, non-volatile memory that sits on the memory bus along with a traditional DRAM. Many papers pointed out that its performance is nothing like a slower DRAM [25, 50]. We briefly discuss its non-uniform, modulated performance findings from previous works.

Small writes result in write amplification. Many state-of-art solutions rely on the performance observation that writes smaller than 256B results in write amplification [5, 25, 50].

Sequential write vs. Random write. Some papers propose that at 256B granularity, random write access to PMEM can be as fast as sequential write access with many threads [5]. However, for a small number of threads, sequential write has a higher bandwidth than random write. *ENTS leverages the performance of sequential write.*

Small hot-spot regions cause high tail latency. Previous work found that overwriting a small region over and over results in a few orders of magnitude higher tail latency [50]. They observe latency spikes regularly, which hints that hardware wear-leveling is triggered when writes stress one specific region. *ENTS avoids overwrites to the same memory region with append-only sequential writes.*

DRAM to PMEM copy scales best. Hildebrand et al. stated in their paper the DRAM to PMEM data movement scales the best as the number of threads increases, and it can reach the PMEM write bandwidth [18]. *ENTS eagerly persists data objects from DRAM to PMEM.*

Segregating DRAM/PMEM Prior works [13] also observed that having a copy of data on both the DRAM and PMEM exhibits high performance. Unlike prior work, which focused on lock-free data structure, ENTS targets FASE transaction programming models.

6.4 Reducing Fences

Various works reduced the fence instructions usage [8, 27, 30, 37, 40, 42]. Nonetheless, many still need it.

(Almost) Fence-less Persist Ordering [35] suggested that NTStore would **likely** persist before a temporal store so that an explicit fence between them may be removed. They extend the x86 persistency model to intentionally defer writebacks of the temporal stores to handle a scenario such as slow NTStore or eagerly evicted CPU cachelines. Unlike their model, ENTS does not need any of their requirements: flush, persistency model extension, intentionally delayed writebacks, and hardware modifications. Furthermore, tuning their model for eADR is non-trivial as they leverage the volatility of the temporal store.

Minimally Ordered Data Structure (MOD) modifies a PMEM shadow object and structurally shares data to reduce the shadowing cost [17]. MOD requires a single fence to order between persisting the shadow object and swapping in the shadow object when updating a group of relevant (pointers are chased from a single data object) purely functional data structures. When updating the isolated data objects within a single failure-atomic section, a few fence instructions are needed for failure-atomicity.

Fence-Free Crash Consistent Design (FFCCD) [47] provides two versions of PMEM concurrent garbage collecting solutions: a software-only single-fence design called SFCCD) and a hardware-assisted fence-free design called FFCCD. SFCCD allows reordering between the data compaction and the compaction status update as long as these two writes are atomic. Upon a failure, the SFCCD recovery code re-executes the `memcpy()` to ensure both writes are complete. FFCCD introduces a new instruction, *relocate*, which records per cacheline persistence state. With such hardware support, the software removes the lingering *sfence* and leverages the recorded persistence state during the recovery. ENTS is similar to FFCCD in that both designs reduce flush and fence instructions during the pre-failure execution but instead rely on a clever recovery for correctness. Unlike FFCCD, ENTS does not need hardware support to track fine-grained persistence. Applying our observation to SFCCD is impossible because it must handle read requests while concurrently performing data compaction. Without the *sfence*, a reader thread may concurrently load data from the relocated memory while the data movement is in progress.

7 CONCLUSION

We observed that hardware memory barriers are implicitly issued by concurrency controls/syscalls and proposed a flush-and-fence-free, log-structured PMEM programming model. By eagerly persisting any modification on DRAM data with a fence-free NTStore, ENTS achieves 1.8×/2.1× performance under ADR/EADR when compared against the Clobber-NVM ([46]) and the industry-standard programming library (PMDK [9]), respectively. We conclude that ENTS exhibits high performance without burdening the programmer to perform flushes and fences manually.

ACKNOWLEDGMENTS

This paper is supported in part by NSF grants 1829524, 1817077, 2011212, and PRISM, one of seven centers in JUMP 2.0 (an SRC program sponsored by DARPA).

REFERENCES

[1] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 111–124. https://doi.org/10.1109/HPCA51647.2021.00019

[2] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-Volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management* (Santa Barbara, CA, USA) *(ISMM 2016)*. Association for Computing Machinery, New York, NY, USA, 55–67. https://doi.org/10.1145/2926697.2926704

[3] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[4] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. https://doi.org/10.14778/2735479.2735483

[5] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1077–1091. https://doi.org/10.1145/3373376.3378515

[6] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS '11)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[7] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 441–454. https://doi.org/10.1145/3297858.3304046

[8] 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* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[9] Intel Corporation. 2017. Persistent Memory Development Kit. http://pmem.io/pmdk.

[10] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. 2020. Lazy Release Persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1173–1186. https://doi.org/10.1145/3373376.3378481

[11] Kshitij Doshi, Ellis Giles, and Peter Varman. 2016. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 77–89. https://doi.org/10.1109/HPCA.2016.7446055

[12] Kshitij Doshi and Peter Varman. 2012. WrAP: Managing Byte-Addressable Persistent Memory.

[13] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: Making Lock-Free Data Structures Persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1218–1232. https://doi.org/10.1145/3453483.3454105

[14] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

[15] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenish. 2020. Relaxed Persist Ordering Using Strand Persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.

[16] 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, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 466–478. https://doi.org/10.1145/3352460.3358321

[17] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 775–788. https://doi.org/10.1145/3373376.3378472

[18] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 875–890. https://doi.org/10.1145/3373376.3378465

[19] IEEE and The Open Group. [n. d.]. The Open Group Base Specifications. https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/ https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.

[20] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Chapter 15. https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf, Version December 2016.

[21] Intel. 2021. EADR: New opportunities for persistent memory applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[22] Intel Corporation. 2016. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

[23] Intel Corporation. 2021. Intel® 64 and IA-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[24] 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* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[25] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[26] Jungi Jeong and Changhee Jung. 2021. PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 517–529. https://doi.org/10.1145/3445814.3446698

[27] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 660–671. https://doi.org/10.1145/2830772.2830805

[28] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

[29] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated Persist Ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 58, 13 pages.

[30] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. 2016. Delegated Persist Ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783761

[31] Duncan Laurie. [n. d.]. IPMITOOL. https://github.com/ipmitool/ipmitool. https://github.com/ipmitool/ipmitool

[32] 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. *SIGPLAN Not.* 52, 4 (April 2017), 329–343. https://doi.org/10.1145/3093336.3037714

[33] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 258–270. https://doi.org/10.1109/MICRO.2018.00029

[34] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. *AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture*. Association for Computing Machinery, New York, NY, USA, 757–773. https://doi.org/10.1145/3373376.3378511

[35] Sara Mahdizadeh-Shahri, Seyed Armin Vakil-Ghahani, and Aasheesh Kolli. 2020. (Almost) Fence-less Persist Ordering. *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2020), 539–554.

[36] 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 789–806. https://doi.org/10.1145/3373376.3378456

[37] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[38] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

[39] Pbalcer. [n. d.]. PMDK Allocator. Intel PMEM IO Blog. https://pmem.io/blog/2016/02/persistent-allocator-design-fragmentation https://pmem.io/blog/2016/02/persistent-allocator-design-fragmentation.

[40] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2014. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 265–276.

[41] Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 137 (oct 2018), 27 pages. https://doi.org/10.1145/3276507

[42] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/3079856.3080240

[43] tatp. [n. d.]. Telecom Application Transaction Processing Benchmark. https://tatpbenchmark.sourceforge.net/. https://tatpbenchmark.sourceforge.net/

[44] GCC team. [n. d.]. GCC Atomic Memory Access. https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html. https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html

[45] TPC-C. [n. d.]. TPC-C. https://www.tpc.org/tpcc/. https://www.tpc.org/tpcc/

[46] 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* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 346–359. https://doi.org/10.1145/3445814.3446730

[47] 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* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 274–288. https://doi.org/10.1145/3470496.3527406

[48] Sujay Yadalam, Nisarg Shah, Xiangyao Yu, and Michael Swift. 2022. ASAP: A Speculative Approach to Persistence. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 892–907. https://doi.org/10.1109/HPCA53966.2022.00070

[49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven 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* (Santa Clara, CA, USA) *(FAST'20)*. USENIX Association, USA, 169–182.

[50] 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 *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[51] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744

[52] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo