High-Performance Transactions for Persistent Memories

Aasheesh Kolli
University of Michigan
akolli@umich.edu

Steven Pelley
Snowflake Computing
steven.pelley@snowflake.net

Ali Saidi
ARM
ali.saidi@arm.com

Peter M. Chen
University of Michigan
pmchen@umich.edu

Thomas F. Wenisch
University of Michigan
twenisch@umich.edu

Abstract

Emerging non-volatile memory (NVRAM) technologies offer the durability of disk with the byte-addressability of DRAM. These devices will allow software to access persistent data structures directly in NVRAM using processor loads and stores, however, ensuring consistency of persistent data across power failures and crashes is difficult. Atomic, durable transactions are a widely used abstraction to enforce such consistency. Implementing transactions on NVRAM requires the ability to constrain the order of NVRAM writes, for example, to ensure that a transaction's log record is complete before it is marked committed. Since NVRAM write latencies are expected to be high, minimizing these ordering constraints is critical for achieving high performance. Recent work has proposed programming interfaces to express NVRAM write ordering constraints to hardware so that NVRAM writes may be coalesced and reordered while preserving necessary constraints. Unfortunately, a straightforward implementation of transactions under these interfaces imposes unnecessary constraints. We show how to remove these dependencies through a variety of techniques, notably, deferring commit until after locks are released. We present a comprehensive analysis contrasting two transaction designs across three NVRAM programming interfaces, demonstrating up to 2.5x speedup.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Main memory

Keywords Non-Volatile Memory, Memory Persistency, Recoverability, Transactions

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

ASPLOS '16 April 2–6, 2016, Atlanta, Georgia, USA.
Copyright © 2016 ACM 978-1-4503-4091-5/16/04. . . $15.00
DOI: http://dx.doi.org/10.1145/2872362.2872381

1. Introduction

New types of memory technology are emerging that could significantly change how software handles persistent data. These new technologies, such as phase-change memory, spin-transfer torque MRAM, memristors, and the recently announced Intel/Micron 3D XPoint technology [15], are non-volatile like magnetic disk and flash memory, but offer much faster access latencies than existing non-volatile storage technologies (though likely not as fast as DRAM). Future systems will likely attach these non-volatile memories (NVRAM) directly to the memory bus and allow processors to access them at word granularity via load and store instructions [14, 27].

For compatibility, some software will continue to access persistent data in NVRAM through a block-based, file system interface. However, we expect many programs to access persistent data structures directly in NVRAM using processor loads and stores. Doing so eliminates the need to maintain separate on-disk and in-memory copies of persistent data, and eliminates the overhead of traversing the file system each time persistent data is read or written.

Ensuring that persistent data is consistent despite power failures and crashes is difficult, especially when manipulating complex data structures with fine-grained accesses. One way to ease this difficulty is to access persistent data through atomic, durable transactions, which make groups of updates appear as one atomic unit with respect to failure. Transactions also provide mechanisms for controlling concurrency; in this paper, we assume that transactions use locks in volatile memory for concurrency control. Because of the power and convenience of transactions, many prior works propose providing them on top of NVRAM [8, 22, 33, 34]. We focus our analysis on static transactions (transactions for which lock sets are known a priori), as detailed in Section 3.

Implementing transactions on NVRAM requires the ability to order writes to the NVRAM.[1] For example, in write-ahead logging [24], the commit record for a transaction may only be written to NVRAM after all log records for that transaction have been written; otherwise a failure may cause the system to recover to a state in which only some of the updates are present. We call writes to NVRAM *persists*, and we call constraints on the order of NVRAM writes *persist dependencies*.

Whereas specifying and honoring persist dependencies is essential for correctness, *minimizing* persist dependencies is likely to be essential for performance. Future NVRAM technologies are likely to be slower than DRAM [19], and will only be able to keep up with CPU speeds through techniques such as parallelism, batching, and re-ordering [35], all of which are possible only in the absence of ordering constraints. Pelley and co-authors show that minimizing persist dependencies can make as much as a 30× difference in performance [27].

This paper considers how to implement NVRAM transactions in a way that minimizes persist dependencies. We show that a simple transaction system design enforces many unnecessary persist dependencies and that these dependencies greatly slow down common transaction workloads, that most of the unnecessary dependencies arise as a consequence of performing the commit step of the transaction while locks are held, and how to remove these dependencies by deferring this commit until after locks are released.

Deferring commits leads to the new challenge of correctly ordering the deferred commit operations across all outstanding transactions. To ensure transaction serializability, commit order must match the order in which locks were originally acquired during transaction execution. We show how to minimize persist dependencies through a combination of techniques, including distributed logs [32], deferred commit [10, 16], Lamport/vector clocks to serialize transactions [18], a subtle epoch-based mechanism to recycle log storage, and relaxed persistency models [27].

We implement a transaction system for NVRAM that defers commits, and we measure its performance on simulated NVRAM with a range of device speeds. For two transaction-processing workloads, we find that performance improves by up to 50% under relaxed persistency models [27] and by up to 150% under Intel's recent x86 ISA extensions for NVRAM [14].

We first introduce a brief formalism to enable reasoning about persist dependencies (Section 2). We then derive the minimal persist ordering requirements to implement correct transactions under an idealized programming interface that can specify arbitrary ordering constraints to hardware (Section 3). Such a programming interface is unrealistic; we summarize practical interfaces proposed in the literature and in recent extensions to the x86 ISA (Section 4). We then analyze a straightforward transaction implementation, synchronous commit transactions (SCT), demonstrating how it overconstrains persist ordering (Section 5). Instead, we propose deferred commit transactions (DCT), which can achieve minimal ordering constraints under sufficiently expressive interfaces (Section 6). We evaluate our transaction implementations using the TPCC and TATP transaction processing workloads (Section 7) and end with a survey of related work (Section 8).

[1] Ensuring recoverability without transactions also requires the ability to order writes.

2. Ordering constraints

The ability to order writes is critical to all software that uses persistent storage. Constraining the order that writes persist is essential to ensure consistent recovery, and minimizing these constraints is key to enabling high performance.

Formally, we express an ordering relation over memory events *loads* and *stores*, which we collectively refer to as *accesses*. The term *persist* refers to the act of durably writing a store to persistent memory. We assume persists are performed atomically (with respect to failures) at 8-byte granularity. By "thread", we refer to execution contexts—cores or hardware threads. We use the following notation (adopted from [17]):

- L_a^i: A load from thread i to address a
- S_a^i: A store from thread i to address a
- M_a^i: A load or store by thread i to address a

We reason about two ordering relations over memory events, *volatile memory order* and *persist memory order*. Volatile memory order (VMO) is an ordering relation over all memory events (loads and stores) as prescribed by the memory consistency model for multiprocessors [1]. Persist memory order (PMO) deals with the same events but may have different ordering constraints than VMO. [27] uses the term *persistency model* to describe the types of constraints that hardware allows software to express on the persist memory order.

We denote these ordering relations as:

- $A \leq_v B$: A occurs no later than B in VMO
- $A \leq_p B$: A occurs no later than B in PMO

An ordering relation between stores in PMO implies the corresponding persist actions are ordered; that is,
$A \leq_p B \rightarrow B$ may not persist before A.

3. Transactions under Idealized Ordering

It is not easy for software to express persist dependencies. Simply ordering the instructions that store data to NVRAM is not sufficient: writes to memory (including NVRAM) are cached and may not be written from the CPU cache to NVRAM in the same order the corresponding instructions were executed [2].

In this section, we suppose that software has the ability to specify precisely the persist dependencies for all writes to NVRAM. While this is unrealistically expressive, it provides a useful baseline upon which to build an idealized transaction system that minimizes persist dependencies. In later sections, we implement transactions built on more realistic interfaces and show how a naive implementation of transactions on these interfaces introduces unnecessary ordering constraints.

The most precise way to specify persist dependencies is as a partial order over all persists. This partial order can be expressed as a directed acyclic graph (DAG), where a node in the graph represents a persist, and an edge exists from node A to node B iff the persist represented by node A must occur no later than the persist represented by node B (note that this condition can be satisfied by performing the two persists atomically). In a system with idealized ordering, the software can express a constraint between any two persists, including persists that occur on separate threads.

We next describe how to build a simple transaction system, given the ability to express general partial orders over all NVRAM writes.

3.1 Transaction design

There are many ways to implement transactions [12], with one basic design choice being which version to log of the data being modified in a transaction: the data before the modification (undo logging [8, 11, 21]), the data after the modification (redo logging [33]), or both (e.g., ARIES [24]). In this paper, we implement transactions with undo logging. We believe this design fits well with storing data directly in NVRAM: both committed and uncommitted data are stored in place, so software can always read the most recent data directly from the in-place data structure (assuming appropriate locks are held). In contrast, if transactions are implemented with a redo log, reads of uncommitted data must be intercepted and redirected to the redo log.

We further implement several common optimizations required to achieve high transaction concurrency. We implement per-thread, distributed logs [32, 33], to avoid the scalability constraints of a centralized log. Our undo log records a copy of data (physical undo records) before it is mutated rather than a "synchronous log-and-update" approach (like PMFS [11]), as the latter requires more persist ordering constraints. We leverage checksum-based log entry validation [28] so that non-atomic writes to a log entry can proceed in parallel, but recovery software can deduce whether a log record was fully written without requiring a separate "valid" bit. This optimization eliminates one persist ordering constraint and is similar to the torn-bit optimization in Mnemosyne [33] and eager commit [22]. We assume concurrency control via arbitrarily fine-grain locking—a transaction must hold all required locks before executing (i.e. static transactions). Requiring a transaction to hold all locks before executing implies that all the data that can possibly be modified by the transaction is known a priori. If such knowledge is not available, a program must execute a read phase to identify all regions it might touch and acquire all locks, and then begin execution (similar to the approach used to implement deterministic transactions [29]).

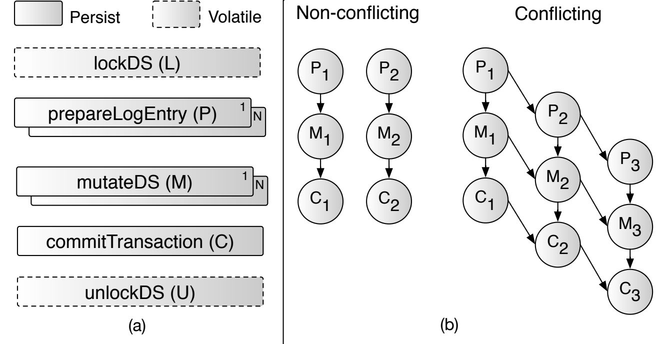


Figure 1. (a) Steps in an undo transaction. (b) Persist dependencies in a transaction sequence.

Figure 1(a) depicts the high-level steps of an undo-logging transaction. Steps outlined in a dotted box modify only volatile memory locations; those outlined in a solid box write to persistent memory. We briefly describe each step:

- **lockDS (L):** Acquire all locks to ensure mutual exclusion of the transaction. Locks are held in volatile memory.
- **prepareLogEntry (P):** Allocate log space and copy the prior state of all data that will be mutated to the log.
- **mutateDS (M):** Modify the data structure in place.
- **commitTransaction (C):** Commit the transaction by marking the undo log entry invalid; the transaction will no longer be undone during recovery.
- **unlockDS (U):** Release all locks acquired by lockDS.

We represent transactions with three persist nodes, corresponding to the three steps that perform durable writes, $prepareLogEntry$ (P), $mutateDS$ (M) and $commitTransaction$ (C).

3.2 Minimal Persist Dependencies

We next analyze the minimal persist dependencies required for correct recovery of an undo-logging transaction. We consider two transactions, T_m and T_n, which acquire lock sets $Locks_m$ and $Locks_n$, respectively. The transactions conflict if their lock sets intersect (i.e., they mutate overlapping data). We require order across conflicting transactions (the order in which they acquire locks); the subscripts indicate this order—in our example, $m < n$. $transactionStep_m$ indicates completion of a particular step in the transaction T_m (and all its associated persists). Recovery correctness requires the following order relationships:

$$prepareLogEntry_m \leq_p mutateDS_m \quad (1)$$

$$mutateDS_m \leq_p commitTransaction_m \quad (2)$$

$\forall(m,n):$
$(unlockDS_m \leq_v lockDS_n) \wedge (Locks_m \cap Locks_n \neq \phi)$

$$prepareLogEntry_m \leq_p prepareLogEntry_n \quad (3)$$

$$mutateDS_m \leq_p mutateDS_n \quad (4)$$

$$commitTransaction_m \leq_p commitTransaction_n \quad (5)$$

- Within one transaction, the log entry must be complete before data structure mutation (Eq. 1), and mutation must be complete before the transaction commits (Eq. 2). These dependencies ensure that any incomplete transaction can be rolled-back during recovery.
- Between conflicting transactions, preparing the log, mutating data, and commit must be ordered (Eqs. 3, 4, 5). These dependencies ensure that: (1) Mutations from conflicting transactions persist in lock-acquisition order (Eq. 4). (2) During recovery, active log entries from conflicting transactions can be undone in the appropriate order (Eqs. 3, and 5). Note that no dependencies exist between non-conflicting transactions.

3.3 Persist critical path analysis

In later sections, we evaluate alternative transaction implementations by comparing their persist dependency critical path to the ideal persist dependency DAG. Conflicting transactions incur additional dependencies that are absent among non-conflicting transactions. Hence, we characterize the critical path under two extreme scenarios, one where all transactions are non-conflicting, and one where all transactions conflict. Figure 1(b) depicts the ideal DAG for conflicting and non-conflicting transaction sequences. Nodes in this figure correspond to the (concurrent) sets of persist operations performed in each transaction step (we omit steps that modify only volatile state). Edges indicate persist dependency between nodes (more precisely, pairwise persist dependencies between all persists represented by each node).

Under each scenario, we assume x transactions are performed, and t threads concurrently execute those transactions. In the non-conflicting scenario, the x transactions all acquire disjoint locks and modify disjoint data. Therefore, there are no persist order dependencies across threads; the critical path is determined solely by persist ordering constraints that arise on a single thread. In this scenario, the ideal persist critical path length is 3—the intra-transaction ordering constraints—independent of x or t.

In the conflicting scenario, we assume all x transactions mutually conflict (they all require a lock in common). Therefore, the persist critical path follows the total order of these x transactions, as established by the order the locks are acquired. In this case, the persist critical path propagates through the commit node of each transaction, resulting in a critical path length of $x + 2$ persist operations. Again, the critical path is independent of the number of threads t.

While persist critical paths for an ideal DAG are quite short, achieving this ideal is difficult with currently proposed programing interfaces, which we summarize next.

4. Memory persistency models

Section 3 supposes that software is able to specify arbitrary ordering relationships among all persists, for example, in the form of a DAG. With this ability, a transaction implementation can minimize the number of persist dependencies and maximize NVRAM performance.

However, it is impractical to expect hardware to allow software to specify arbitrary persist dependencies, as this would require the hardware to track and honor an arbitrary DAG among persists. Instead, hardware will likely provide persist ordering mechanisms similar to those for ordering memory accesses in shared-memory multiprocessors. Industry has already begun following this course [14].

In currently shipping processor architectures, persist dependencies must be enforced either by using a write-through cache or by explicitly flushing individual cache lines (e.g., using the *clflush* instruction on x86). Moreover, these flush operations must be carefully annotated with fences to prevent hardware and compiler reorderings (details appear in [2]). These mechanisms are quite slow because they give up much of the performance benefits of CPU caches. Because cache flushes are so slow, Intel has recently announced extensions to its x86 ISA to optimize cache line flushing [14]. However, these mechanisms tie the ordering required between writes to NVRAM to the ordering required between a write to NVRAM and subsequent CPU instructions (Chidambaram, et al. describe this as the distinction between ordering and durability [6]).

Researchers have proposed other means to express persist dependencies. Condit and co-authors propose an *epoch barrier*, which ensures writes before the barrier are ordered before writes after the barrier [9]. Pelley and co-authors liken the problem of ordering persists to the problem of ordering memory accesses in a multiprocessor [27]. Just as there is a design space for multiprocessor memory consistency models, Pelley lays out a design space for NVRAM memory *persistency* models. We use *Total Store Order* as the underlying consistency model in this paper. We briefly summarize four persistency models, on which we build our transaction implementations.

4.1 Strict persistency

Under strict persistency, PMO is identical to VMO. So, for any two stores ordered by the consistency model, the corresponding persists are also ordered. Formally,

$$M_a^i \leq_v M_b^j \leftrightarrow M_a^i \leq_p M_b^j \quad (6)$$

Whereas strict persistency is the most intuitive persistency model, it is not the best performing. By ordering persists per VMO, strict persistency enforces orderings typically not

required for recovery correctness [27]. Thus, researchers have proposed more relaxed persistency models, in which PMO may have fewer ordering constraints than VMO.

4.2 Epoch persistency

The *epoch persistency model* introduces a new memory event, the "persist barrier" (different from memory consistency barriers). We denote persist barriers issued by thread i as PB^i. Under epoch persistency, any two memory accesses on the same thread that are separated by a persist barrier in VMO are ordered in PMO.

$$M_a^i \leq_v PB^i \leq_v M_b^i \rightarrow M_a^i \leq_p M_b^i \tag{7}$$

Persist barriers separate a thread's execution into ordered epochs (persists within an epoch are concurrent). While persist barriers order persists from one thread, epoch persistency relies on another property, *strong persist atomicity*, to order persists from different threads.

Strong persist atomicity: Memory consistency models often guarantee that stores to the same address by different processors are serialized (this is called *store atomicity*). Pelley argues persistency models should similarly provide *strong persist atomicity* (SPA), to preclude non-intuitive behavior, such as recovering to states unreachable under fault-free execution [27]. SPA requires that conflicting accesses (accesses to the same address, at least one being a store) must persist in the order they executed.

$$\begin{aligned} S_a^i \leq_v M_a^j \rightarrow S_a^i \leq_p M_a^j \\ M_a^i \leq_v S_a^j \rightarrow M_a^i \leq_p S_a^j \end{aligned} \tag{8}$$

4.3 Strand persistency

Strand persistency divides program execution into *strands*. Strands are logically independent segments of execution that happen to execute in the same thread. Strands are separated by the *new strand* (NS) memory event. New strand events from thread i are denoted as NS^i. The new strand event clears all prior PMO constraints from prior instructions, effectively making each strand behave as if it were a separate thread (with respect to persistency). Memory accesses within a strand are ordered using persist barriers (Eq. 7). Under strand persistency, two memory accesses on the same thread separated by a persist barrier are ordered in PMO only if there is no intervening strand barrier. Memory accesses across strands continue to be ordered via SPA (Eq. 8).

$$\begin{aligned} (M_a^i \leq_v PB^i \leq_v M_b^i) \wedge (\nexists NS^i : M_a^i \leq_v NS^i \leq_v M_b^i) \\ \rightarrow M_a^i \leq_p M_b^i \end{aligned} \tag{9}$$

4.4 Eager sync

In addition to the persistency models proposed by Pelley, we also consider *eager sync*, our attempt to formalize the persistency model implied by Intel's recent x86 ISA extensions to optimize NVRAM performance [14]. We briefly describe these new instructions:

- $CLWB$: Requests write back of modified cache line to memory; the cache line may be retained in a clean state.
- $PCOMMIT$: Ensures that stores that have been accepted to memory are persistent.

Using these two instructions, stores on one thread to addresses A and B can be guaranteed to persist in the order $S_A^i \leq_p S_B^i$, using the following pseudo-code:

```
st A; CLWB A; SFENCE; PCOMMIT; SFENCE; st B;
```

We use the term "sync barrier" to refer to the code sequence $SFENCE; PCOMMIT; SFENCE$. A sync barrier issued by thread i will be denoted as SB^i. The first $SFENCE$ orders the $PCOMMIT$ with the earlier stores and $CLWB$s, while the second orders the younger stores with the $PCOMMIT$. A sync barrier differs from a persist barrier under epoch and strand persistency in two ways: (1) The second $SFENCE$ ensures that a younger store will *not* be globally visible until all stores older than the $PCOMMIT$ become persistent. In contrast, a persist barrier does not affect the global visibility of subsequent stores, it only orders the corresponding persists. (2) The $PCOMMIT$ persists only those stores that have been accepted to memory (e.g., using $CLWB$); a persist barrier orders the persists in PMO for all stores that precede the persist barrier in VMO.

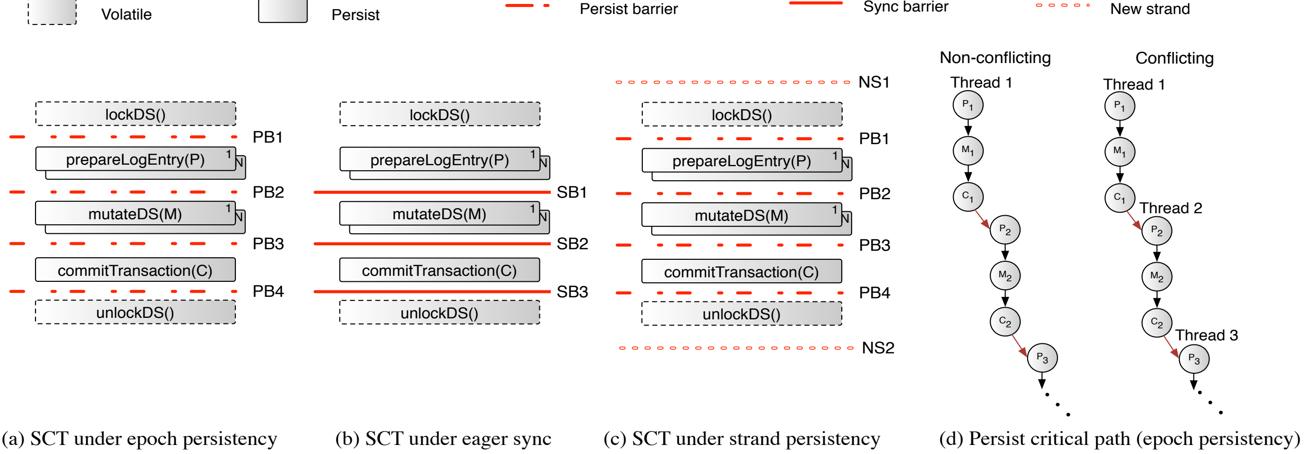
Sync barriers affect the visibility of subsequent stores, with two important implications:

- Delaying store visibility until prior stores become persistent will likely add stalls to thread execution, especially because NVRAM persists may take 100s of nanoseconds [19] and a $PCOMMIT$ must persist all stores that have been accepted to memory.
- Since global visibility of a store can be delayed until prior stores have become persistent, the visibility of a store can act as a signal that all earlier stores are persistent. Astonishingly, such stalling of volatile events for persist events, can, in some cases, lead to fewer persist dependencies and better performance than epoch persistency (as we show in Section 6).

5. Synchronous commit transactions (SCT)

Section 3 showed how to implement transactions under an idealized programming model allowing arbitrary persist dependencies. We next examine how to implement transactions using more realistic mechanisms.

We first discuss an intuitive transaction implementation, which we call synchronous commit transactions (SCT). However, as we will show, SCT enforces unnecessary persist dependencies and overconstrains the persist critical path. Below, we describe and analyze SCT under epoch, eager sync, and strand persistency (we omit analysis under strict persistency to save space; all our designs will work under strict persistency).



(a) SCT under epoch persistency (b) SCT under eager sync (c) SCT under strand persistency (d) Persist critical path (epoch persistency)

Figure 2. Synchronous-commit transactions under epoch persistency, eager sync and strand persistency. The red arrows in (d) represent the unnecessary dependencies enforced when compared to the minimal dependencies shown in Figure 1b.

5.1 SCT under Epoch Persistency

Epoch persistency enforces intra-thread persist dependencies via persist barriers, and inter-thread dependencies (for conflicting transactions) via persist barriers and SPA (Eq. 8). Figure 2(a) depicts a synchronous-commit transaction annotated with the four persist barriers required for correctness.

Intra-transaction dependencies: $PB2$ and $PB3$ ensure proper intra-transaction ordering of $prepareLogEntry$, $mutateDS$, and $commitTransaction$ (Eqs. 1 and 2).

Inter-transaction dependencies: Conflicting transactions (T_m and T_n) are synchronized through the common locks in their lock sets and hence through $unlockDS_m$ and $lockDS_n$. Since T_m happens in VMO before T_n, from SPA (Eq. 8), we have:

$$unlockDS_m \leq_v lockDS_n \rightarrow unlockDS_m \leq_p lockDS_n$$

The VMO of $prepareLogEntry$, $PB2$ (or $PB3$ or $PB4$), and $unlockDS$ in T_m imply:

$$prepareLogEntry_m \leq_p unlockDS_m$$

The VMO of $lockDS$, $PB1$, and $prepareLogEntry$ in T_n imply:

$$lockDS_n \leq_p prepareLogEntry_n$$

Applying transitivity to the above three equations, we observe that conflicting transactions prepare their log entries in order, satisfying Eq. 3. It is important to note that $PB1$ is critical to ensuring the correct order. Similarly, the VMO of $lockDS$, $PB1$ (or $PB2$), $mutateDS$, $PB3$ (or $PB4$), and $unlockDS$ ensures that conflicting transactions mutate the data structure in order, satisfying Eq. 4. VMO of $lockDS$, $PB1$ (or $PB2$ or $PB3$), $commitTransaction$, $PB4$, and $unlockDS$ ensure that conflicting transactions commit in order, satisfying Eq. 5.

Thus, the four persist barriers in Figure 2(a) are necessary and sufficient to ensure transactional persist ordering requirements. Unfortunately, these four persist barriers create a persist critical path longer than the path that would be possible had the software been able to specify the precise dependencies between all persists (Section 3).

From the VMO of $commitTransaction$, $PB4$, and $unlockDS$ in T_m and Eq. 7, we have:

$$commitTransaction_m \leq_p unlockDS_m$$

Similarly, VMO of $lockDS$, $PB1$, and $prepareLogEntry$ in T_n implies:

$$lockDS_n \leq_p prepareLogEntry_n$$

We have already shown:

$$unlockDS_m \leq_p lockDS_n$$

Applying transitivity to the above three equations, we have:

$$commitTransaction_m \leq_p prepareLogEntry_n$$

So, under epoch persistency, conflicting transactions are serialized. Moreover, transactions on the same thread are always serialized, even if they do not conflict. Figure 2(d) shows the persist critical path under epoch persistency: $3x$ for conflicting transactions and $3(x/t)$ for non-conflicting transactions. Both are longer than the minimal critical path (Figure 1(b)). Whereas SCT under epoch persistency is simple and intuitive, performing all steps of a transaction while holding locks overconstrains the persist dependency graph and lengthens the persist critical path.

5.2 SCT under Eager Sync

Eager sync enforces both intra-thread and inter-thread (for conflicting transactions) persist dependencies via sync barriers. Figure 2(b) depicts a synchronous-commit transaction annotated with the three sync barriers required for correctness. We also assume that all the $CLWBs$ required to be

issued before the sync barriers are issued along with the stores in the functions $prepareLogEntry$, $mutateDS$ and $commitTransaction$.

Intra-transaction dependencies: $SB1$ and $SB2$ ensure correct intra-transaction ordering of $prepareLogEntry$, $mutateDS$, and $commitTransaction$, satisfying Eqs. 1,2.

Inter-transaction dependencies: We again consider conflicting transactions T_m and T_n. $SB3$ ensures $unlockDS_m$ is not globally visible until $commitTransaction_m$ persists to NVRAM. $prepareLogEntry_n$ cannot be executed until T_n acquires its locks ($lockDS_n$), which happens after $unlockDS_m$ becomes globally visible. By stalling the global visibility of $unlockDS_m$ until $commitTransaction_m$ persists (because of $SB3$), we ensure that:

$$commitTransaction_m \leq_p prepareLogEntry_n$$

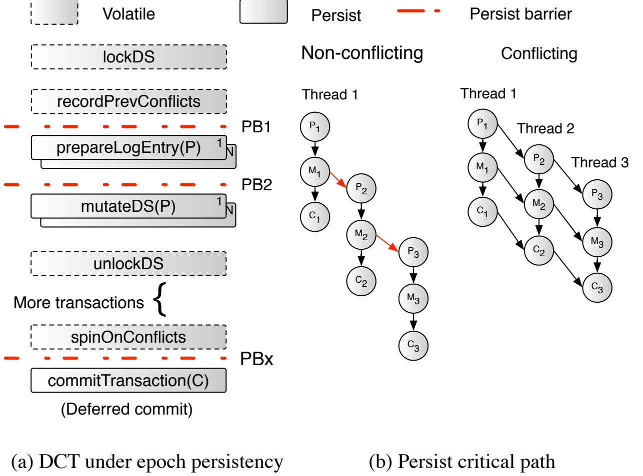
It is important to note that a sync barrier between $LockDS$ and $prepareLogEntry$ is not required to achieve the above dependency. The above dependency is the same (over-constraining) dependency incurred under epoch persistency, which serializes all conflicting transactions. $SB3$ also enforces that non-conflicting transactions within the same thread are serialized (as under epoch persistency). SCT under eager sync enforces the same ordering constraints as SCT under epoch persistency, resulting in the same persist critical path (Figure 2(d)).

5.3 SCT under Strand Persistency

Strand persistency makes it possible to remove unnecessary persist dependencies between transactions on the same thread (left graph of Figure 2(d)) by placing the transactions on different *strands*. Our implementation of SCT is shown in Figure 2(c). We start and end every transaction on a new strand ($NS1$, $NS2$ in Figure 2(c)). As a result, each transaction behaves as if on its own logical thread (from the perspective of the persistent memory, from Eq. 9). Such a design removes the dependence between successive non-conflicting transactions on the same thread. Conflicting transactions continue to be ordered due to the dependencies caused by the lock/unlock operations (as under epoch persistency).

It is important to note that in the SCT design for strand persistency (Figure 2(c)), $NS1$ ensures that each transaction starts on a new strand, and $NS2$ ensures that memory events executed after the transaction (but prior to the next transaction), don't end up serializing transactions on the same thread. For example, transaction systems may perform some book-keeping (say, update transaction count) at the end of every transaction. Without $NS2$, such bookkeeping could end up causing conflicts between otherwise non-conflicting transactions.

To achieve high concurrency, our SCT implementation uses per-thread (distributed) logs. In practice, log space is limited, and must ultimately be recycled. As a consequence, transactions that share no locks may nonetheless conflict if they reuse the same log space. We enforce the necessary ordering by adding a lock for the log entry to the transaction's lock sets.



(a) DCT under epoch persistency (b) Persist critical path

Figure 3. Deferred commit transactions under epoch persistency and the resulting persist ordering constraints.

Under ideal implementations of strand persistency, the achievable persist concurrency is limited only by the available log space. In practice, we expect future systems to limit strand concurrency. In a system with t threads and s strands, the maximum concurrency under strand persistency is similar to a system with $s \times t$ threads under epoch persistency. Under strand persistency, the SCT persist critical path for non-conflicting transactions improves to $3(x/st)$ (the persist critical path for conflicting transactions remains $3x$). Whereas strand persistency improves the SCT persist critical path, it remains longer than the theoretical minimum.

6. Deferred commit transactions (DCT)

SCT generates longer critical paths than needed when implemented on realistic persistency models. In this section, we describe deferred commit transactions (DCT), which generate shorter critical paths than SCT.

The key idea in DCT is for a transaction to release locks after mutating the data structure and to defer commit until later. This idea has been explored as a mechanism for managing lock contention for transaction systems with a centralized log [16]. We use this idea to break the persist order dependence between $commitTransaction$ and $prepareLogEntry$ of consecutive conflicting transactions. However, performing the commit after the lock release implies that the persists from $commitTransaction$ are no longer synchronized by the respective locks and could result in conflicting transactions committing out of order. [2] To ensure that conflicting transactions commit in order (Eq. 5) we modify transactions to explicitly track (in volatile memory)

[2] This is not a problem with a centralized log, as they are serialized by the lock for the log.

their predecessor conflicting transactions and commit after all predecessors have committed. Next, we describe DCT implementations under the three persistency models.

6.1 DCT under Epoch Persistency

Figure 3(a) shows the implementation of DCT and the associated "deferred-commit" block.

Intra-transaction dependencies: Persist barrier $PB2$ helps satisfy Eq. 1 by guaranteeing that $prepareLogEntry$ is ordered before $mutateDS$. The commit-after-mutate rule (Eq. 2) is ensured by PBx (a barrier from a subsequent transaction).

Inter-transaction dependencies: For conflicting transactions T_m and T_n, the persist barriers $PB1$ and $PB2$, along with the SPA guarantees of $unlockDS_m$ and $locksDS_n$, ensure that the log entries are prepared in order, satisfying Eq. 3. SPA (Eq. 8) of the conflicting regions of the data structure ensure that Eq. 4 is satisfied. DCTs need to explicitly track prior conflicting transactions to ensure the commit-in-order rule (Eq. 5). We achieve this order by having the transaction $spinOnConflict$ (conflicts are recorded in the log entry) after the lock release and then $commitTransaction$ following a persist barrier PBx. It is important to note that, instead of having a designated barrier to order the commit, we rely on a barrier from a subsequent transaction. As a result, the $commitTransaction$ step may occur concurrently with persists from a subsequent transaction and does not add to the persist critical path. Next, we describe the challenges that arise from deferring commits and the bookkeeping required to address them.

6.1.1 Inferring undo order during recovery

By allowing transaction commits to be deferred, we can arrive at a state where multiple conflicting uncommitted transactions must apply undo log entries at recovery. The recovery protocol must infer the order of these log entries and perform the undo operations in reverse order. As we use distributed logs, deducing this order is non-trivial. Mnemosyne [33] uses a single global atomic counter to assign each new transaction an incrementing global timestamp (log entries can be undone in the decreasing order of timestamps). However, such an approach implies that all transactions conflict (they all update the global counter), and results in an artificially conflated persist critical path. One might consider recording a timestamp in each log entry, but reliably ordering nearly concurrent events via wall-clock timestamps is challenging, especially if execution is distributed over multiple cores/chips.

Since we only need to order log entries for conflicting transactions, we extend all locks to contain logical time stamps (i.e., Lamport clocks [18]). When a transaction acquires a lock, it records and increments the current lock timestamp, ensuring subsequent conflicting transactions will see a higher timestamp. Timestamps are logged in the new $recordPrevConflicts$ function, shown in Figure 3(a). If a transaction acquires multiple locks, all of their timestamps must be recorded. Recovery uses these timestamps to deduce the correct undo order.

6.1.2 Enforcing correct commit order

To ensure correct recovery, conflicting transactions must commit in order. DCT requires an explicit software mechanism to track and enforce this order. We extend each lock with a pointer to the log entry of the last transaction to acquire that lock. When a transaction acquires all of the locks in its lock set, it records the pointers to previous conflicting transactions (one per lock) in volatile memory, shown as $recordPrevConflicts$ in Figure 3(a). Then, it records a pointer to its own log entry in each lock.

At commit, a transaction must verify that preceding conflicting transactions have committed. Using the recorded pointers, it examines each preceding log entry for a commit marker, spinning until all are set ($spinOnConflicts$ in Figure 3(a)). However, if a log entry is recycled, its commit marker is now stale. Along with the commit marker, $recordPrevConflicts$ records a log generation number associated with every log entry. The log generation number is incremented if the log entry is recycled. The combination of the commit marker and the log generation number is used to deduce whether an earlier transaction has committed.

Once all of the conflicting log entries are committed, the transaction may commit ($commitTransaction$ in Figure 3(a)). (Note that, rather than simply spinning, an intelligent transaction manager could instead further defer commit and execute additional transactions on this thread). The spin loop prior to commit orders conflicting transactions in VMO. A persist barrier is required between $spinOnConflicts$ and $commitTransaction$ (PBx) to ensure the conflicting transaction commits are also ordered in PMO.

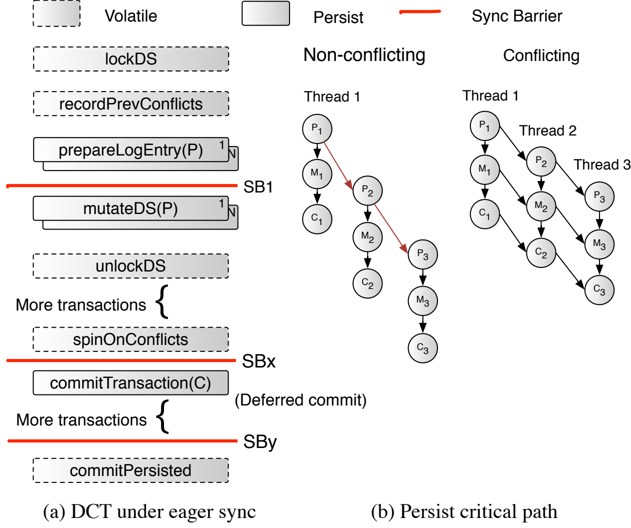
6.1.3 Persist critical path analysis

Figure 3(b) shows the persist critical path for DCT under epoch persistency. DCT succeeds in matching the critical path length of the ideal dependence DAG for conflicting transactions as derived in Section 3. For transactions on a single thread, it reduces the critical path by allowing commit operations to be batched.

For non-conflicting transactions on the same thread, the $prepareLogEntry$ and $mutateDS$ steps remain (unnecessarily) serialized. The $commitTransaction$ step overlaps with the $prepareLogEntry$ step of the subsequent transaction. Hence, the non-conflicting persist critical path length is $(2(x/t)+1)$. For conflicting transactions, the persist critical path traverses the $commitTransaction$ step of each transaction, and its path length is $x+2$.

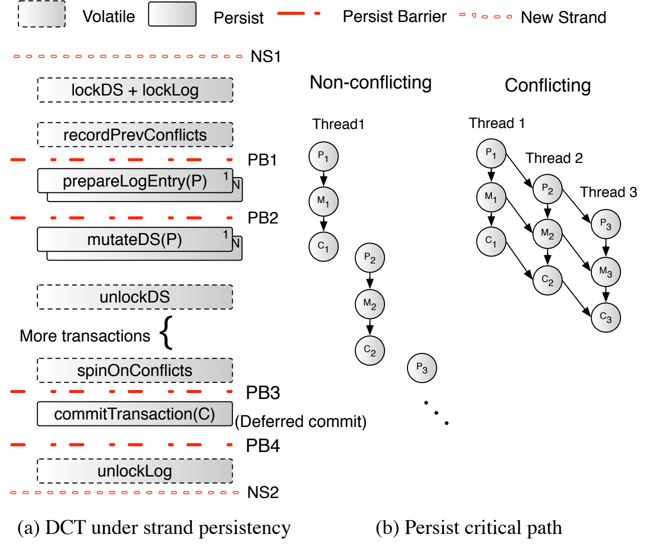
6.2 DCT under Eager Sync

The implementation of DCT under the eager sync persistency model is shown in Figure 4(a). While similar to the DCT implementation under epoch persistency, we require



(a) DCT under eager sync (b) Persist critical path

Figure 4. Deferred commit transactions under eager sync and the resulting persist ordering constraints.



(a) DCT under strand persistency (b) Persist critical path

Figure 5. Deferred commit transactions under strand persistency and the resulting persist ordering constraints.

some subtle changes to account for the differences between a persist barrier and a sync barrier detailed in Section 4.2. It is important to note that we require only one sync barrier within the transaction, rather than the two persist barriers required for DCT under epoch persistency.

Intra-transaction dependencies: The sync barrier $SB1$, ensures order between $prepareLogEntry$ and $mutateDS$, satisfying Eq. 1. The sync barrier Sbx (which belongs to a subsequent transaction) ensures that $mutateDS$ and $commitTransaction$ are ordered correctly, satisfying Eq. 2.

Inter-transaction dependencies: We discuss inter-transaction dependencies using two conflicting transactions T_m and T_n. Within T_m, $SB1$, ensures that $unlockDS_m$ doesn't become globally visible until $prepareLogEntry_m$ becomes persistent. In transaction T_n, $prepareLogEntry_n$ cannot be executed before the locks are acquired using $lockDS_n$. However, $lockDS_n$ cannot be completed until $unlockDS_m$ becomes globally visible. Transitively, $SB1$, ensures that log entries are prepared in order, satisfying Eq. 3. Cache coherence ensures that at any given time, only the latest values of any conflicting regions of the data structure persist, satisfying Eq. 4.

Since SPA (Eq. 8) is not provided under eager sync, we cannot use the same coding pattern as used in the epoch persistency DCT implementation to ensure conflicting transactions commit in order. Instead, we have a $commitPersisted$ bit associated with every log entry, which is set after $commitTransaction_m$ is guaranteed to be have persisted (ensured by SBy in Figure 4(a)). We modify the $spinOnConflict$ function to spin on the $commitPersisted$ bits of conflicting transactions, rather than the log entries. Once a transaction observes that the $commitPersisted$ bit of earlier conflicting transactions have been set, it can be committed and be certain that the correct commit order has been followed.

It is important to note that we do not need dedicated sync barriers, SBx and SBy, for every transaction. We instead rely on sync barriers from a subsequent transaction, implying that both $mutateDS$ and $commitTransaction$ are persisted concurrently with later transactions. So, only the persists belonging to $prepareLogEntry$ fall on the persist critical path on a single thread, as depicted in Figure 4(b). For non-conflicting transactions, the persist critical path traverses all the $prepareLogEntry$ steps of each transaction executed on one thread and is $x/t + 2$. For the conflicting case, the persist critical path traverses the $commitTransaction$ step of all the transactions and is $x + 2$. Note that DCT under eager sync incurs a shorter persist critical path than under epoch persistency for non-conflicting transactions, whereas they exhibit the same persist critical path for conflicting transactions.

Discussion: DCT under eager sync and Mnemosyne (asynchronous mode) [33] are similar in that each transaction may add at most one persist epoch delay to the execution critical path. Whereas DCT amortizes the cost of $mutateDS$ and $commitTransaction$ over subsequent transactions on the same thread, Mnemosyne offloads log truncation to a separate helper thread.

6.3 DCT under Strand persistency

Figure 5(a) shows our implementation of DCT under strand persistency. As with the SCT implementation under strand persistency, we expose additional persist concurrency by placing each transaction on a new strand, removing the dependencies between non-conflicting transactions on the same thread. Similar to SCT, we introduce a log entry lock to a transaction's lock set, so that transactions which

	Non-conflicting		Conflicting	
Persistency Model	**SCT**	**DCT**	**SCT**	**DCT**
Epoch	$3x/t$	$2x/t+1$	$3x$	$x+2$
Eager sync	$3x/t$	$x/t+2$	$3x$	$x+2$
Strand	$3x/st$	$3x/st$	$3x$	$x+2$

Notation: x- total transactions, t-threads, s-strands/thread

Table 1. Summary of persist critical path lengths.

conflict on a log entry are serialized. The log entry lock is acquired along with all the other locks in a transaction's lock set. However, the log entry is only released after $commitTransaction$, serializing transactions that share log space. Figure 5(b) shows the persist critical paths for the conflicting and non-conflicting scenarios. In the conflicting case, as under epoch persistency, the persist critical path passes through the $commitTransaction$ step of each transaction, leading to the ideal persist critical path length of $x+2$ edges. In the non-conflicting case, the persist dependency critical path improves, but may still fall short of the ideal DAG if the number of strands supported in hardware is limited. The persist critical path for non-conflicting transactions goes through transactions which share log space and is $3x/st$ where t is the number of threads and s the number of strands per thread (similar to SCT under strand). With support for at least two strands per thread, DCT under strand persistency outperforms DCT under epoch persistency.

Table 1 summarizes the critical paths for SCT and DCT under various persistency models and workloads.

7. Evaluation

We evaluate transactional systems implementing both SCT and DCT to examine their performance trade-off as a function of persist latency for two transaction processing workloads. In general, we expect DCT to have slower volatile execution performance, due to the bookkeeping overheads required to order commits. However, as persist latency increases, it rapidly becomes the performance bottleneck, and DCT overtakes SCT. As the NVRAM programming interface remains unclear, we also compare the performance achieved under different persistency models for both SCT and DCT. Strict persistency performs much worse than all other persistency models, so we omit it from the discussion.

7.1 Methodology

We implement our transactional systems as a C++ library providing a simple API comprising only three entry points: `beginTransaction()`, `prepareLogEntry()`, and `endTransaction()`. The system manages bookkeeping, log serialization, commit ordering, and inserting the necessary barriers to enforce persist dependencies.

Because memory-bus-attached NVRAM devices are not yet available, we use a region of DRAM as a proxy. We execute our workloads writing persistent data to DRAM to establish their volatile execution performance. We then re-execute the workloads with lightweight pin instrumentation [23] to record all persist operations and barriers. From these traces, we construct the persist critical path (taking into account ordering within and across threads). We assume 8-byte atomic persists.

Under epoch and strand persistency, overall throughput is limited by the slower of volatile execution and the latency to drain all persists. However, in the case of eager sync, overall throughput is limited by volatile execution, which includes the stalls associated with executing the sync barriers. The overhead of a sync barrier only includes the latency to make the stores persistent and does not include the costs associated with issuing and executing the corresponding $CLWB$s.

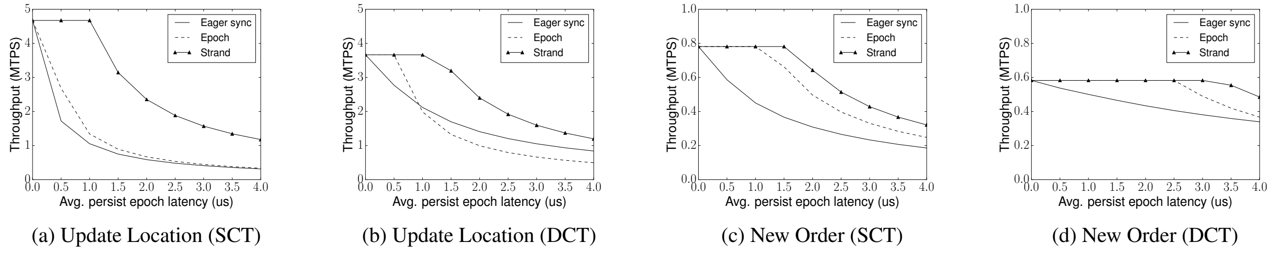
As the hardware characteristics, raw device latency, and scheduling limitations of a practical persistent memory system are as yet unknown, we vary our assumption for persist performance and report the resulting throughput. Cumulative persist latency is determined by how fast, on average, an epoch of persists can drain subject to queueing, scheduling, device-level concurrency, and coalescing effects. We abstract these effects as a single average latency per persist epoch (i.e., latency per dependency edge in the critical path). As we expect persist throughput to be the performance bottleneck when transactions are short, load on the persistent memory system will be high and queueing delays substantial. Hence, the average persist epoch latency is likely a small integer multiple of the NVRAM device latency, we study the the range of 0-4μs.

We perform experiments on an Intel Xeon E5645 processor (2.4GHz). We analyze throughput for transactions selected from two widely studied transaction processing workloads. We study the *New Order* transaction from TPCC [30], which is its most frequent write transaction. A New Order transaction simulates a customer buying different items from a local warehouse. The transaction is write-intensive and requires atomic updates to several tables. We also study the *Update Location* transaction from TATP [26], a benchmark that models a mobile carrier database. Update Location records the hand-off of a user from one cell tower to another. In contrast to New Order, Update Location transactions are much shorter, updating a small record in a single table. We execute workloads for 10M transactions running on four threads. We assume four strands per thread under strand persistency.

7.2 Performance analysis

Figure 6 contrasts SCT and DCT performance across workloads, persistency models, and average persist epoch latencies depicting throughput (millions of transactions per sec (MTPS)) versus persist epoch latency (micro seconds).

Each performance result with epoch and strand persistency (Figure 6) comprises a flat region, followed by a curve where throughput rapidly falls off. In the flat regions, where average persist epoch latency is low, overall throughput is



(a) Update Location (SCT) (b) Update Location (DCT) (c) New Order (SCT) (d) New Order (DCT)

Figure 6. SCT and DCT performance for Update Location and New Order under various persistency models.

limited by volatile execution. At the knee, which we call the "break-even" latency, volatile execution and the persist critical path are equal. Higher break-even latency implies tolerance for slower NVRAM technologies. Performance then drops off rapidly as average persist epoch latency increases and asymptotically reaches zero. In contrast, for eager sync, since sync barriers cause stalls in volatile execution, performance begins to drop-off at the first non-zero average persist epoch latency.

Volatile execution performance of SCT exceeds DCT. As expected, the additional bookkeeping required to implement DCT penalizes volatile execution speed—SCT transactions are faster than DCT transactions (if persist epoch latency is neglected) by 20%, 25% for Update Location and New Order respectively.

SCT performance across persistency models: Figures 6(a) and 6(c) show the performance of SCT, for Update Location and New Order, for different persistency models. SCT under eager sync always performs worse than under epoch persistency. This behavior is to be expected as SCT exhibits similar persist critical paths under epoch persistency and eager sync. With similar persist critical paths, the performance under epoch persistency is always better than under eager sync. Under epoch persistency, performance is determined by the slower of volatile execution and time taken to drain the persists. However, in the case of eager sync, the time taken to drain persists (stalls due to sync barriers), slows volatile execution.

Also, as expected, SCT performs better under strand persistency than under epoch persistency, due to a shorter persist critical path. Hence, SCT performs best under strand persistency and the worst under eager sync.

DCT performance across persistency models: The performance trade-offs for DCT are more complex. Figure 6(b) shows that the performance of DCT under epoch persistency is worse than under eager sync above 1μs persist latency. DCT incurs a longer persist critical path under epoch persistency than under eager sync, especially for workloads where transactions rarely conflict, like Update Location. Hence, beyond the break-even latency, the performance under epoch persistency declines faster than under eager sync.

	Update Location	New order
Epoch persistency	0.5	2
Eager sync	0.5	1
Strand persistency	1.5	2.5

Table 2. The average persist epoch latency (in μs), at which DCT breaks even with SCT.

For New Order (Figure 6(d)), we see that DCT performs better under epoch persistency than under eager sync. This behavior is caused by multiple factors: (1) The break-even latency for epoch persistency is 2.5μs, so epoch persistency performance degrades only for persist latencies above 2.5μs. (2) New Order has more conflicting transactions than Update Location, so the difference in persist critical path between epoch and eager sync is smaller. (3) The crossover point at which eager sync begins outperforming epoch persistency lies beyond 4μs, which is not shown in the graphs. It is not clear that a memory technology that incurs more than 4μs average persist epoch latency is viable as a main memory.

As expected, DCT under strand persistency performs best for both workloads (Figures 6(b) and 6(d)) as the persist critical path under strand persistency is the shortest.

SCT vs. DCT across persistency models: The performance trade-off (for all persistency models) between SCT and DCT depends upon two competing factors: (1) the better volatile performance of SCT, and (2) the shorter persist critical paths of DCT. As a result, for lower average persist epoch latencies, SCT performs better, but as latency increases, DCT outperforms SCT by up to 50% under epoch and strand persistency and 150% under eager sync.

In Table 2, we summarize the average persist epoch latency, where SCT and DCT provide the same performance, under each persistency model. Table 2 indicates: (1) DCT breaks-even with SCT at higher latencies for New Order than Update Location. New Order is a larger transaction, hiding longer persist delays under volatile execution. (2) Strand persistency exhibits the highest SCT-DCT break-even latencies, as it incurs the smallest difference in persist critical path between DCT and SCT.

8. Related work

The emergence of new persistent memory technologies has spurred research in many areas of computer science, including file systems [4, 9, 11], databases [5, 7, 13, 21, 32], persistent data structures [8, 33], and concurrent programming [3].

Several systems share our goal of providing a transaction interface to persistent memory. NV-Heaps [8] provides a persistent object system with transactional semantics that prevents persistence-related pointer and memory allocation errors. Mnemosyne [33] allows programmers to declare or allocate persistent data and write this data through special instructions or via transactions. Rio Vista [21] provides transactions on top of flat memory regions.

Prior systems have generally not sought to optimize concurrency of writes to persistent memory. For example, Rio Vista assumes persistent memory is fast enough to not require concurrent accesses [21]. NV-Heaps uses epoch barriers to order persistent writes and assumes that memory accesses execute serially [8]. Mnemosyne uses cache-flush operations to order updates to persistent memory [33].

Unlike these systems, our paper focuses on maximizing the concurrency of writes to persistent memory by reducing ordering constraints between persistent memory accesses. We believe that freeing the underlying persistent memory system to reorder, parallelize, and combine writes will be essential to supporting high-performance, transaction-oriented workloads. To our knowledge, our work is the first to explore the implications of various recently proposed persistency models on transaction software.

Recent work by Lu and co-authors shares our goal of reducing ordering constraints among persistent writes [22]. Their system distributes the commit status of a transaction among the data blocks to eliminate an ordering constraint within a transaction (similar to the torn bit in Mnemosyne [33]), and uses hardware support (multi-versioned CPU cache and transaction IDs) to enable conflicting transactions to persist out of order. Their techniques are complementary to the ones we propose for reducing ordering constraints. In addition, their system assumes that flushing is required to guarantee ordering (as in eager sync), whereas we explore other memory persistency models.

Our work builds on prior proposals to allow software to communicate ordering dependencies among writes to persistent memory. In shipping systems, order can be enforced by flushing persistent writes from the CPU cache to memory (e.g., via write-through caches or `clflush` instructions) and then issuing a memory barrier (e.g., `mfence`) [31]. However, flushing data to persistent storage is not necessarily the best way to ensure the order in which data is made durable [6]. To relax ordering requirements, Condit and co-authors propose using *epoch barriers* to ensure an ordering between writes before and after the barrier [9]. Pelley and co-authors expand this into a design space for memory persistency models [27].

Others propose hardware support to increase the apparent speed of persistent memory by adding a nonvolatile CPU cache [34] or by assuming sufficient residual power to complete all pending writes [25]. Reducing persist latency makes it less important to allow concurrent writes to persistent memory. Our work makes the more conservative assumption that data must be written to the main persistent memory to be considered durable. Transactions can also be accelerated via other hardware support for persistent memory, such as editable atomic writes [7].

9. Conclusions

New non-volatile memory technologies make it possible to store persistent data directly in memory. Achieving the full performance benefits of doing so requires minimizing the constraints on the order of writes to NVRAM. In this paper, we show how to design transaction systems that specify and communicate these constraints to hardware in a way that reduces the dependencies between NVRAM writes. Our DCT transaction design reduces the persist critical path and improves performance by up to 50% under epoch and strand persistency and up to 150% under eager sync.

Acknowledgments

We would like to thank the anonymous reviewers for their valuble feedback. This work was supported by the National Science Foundation under the award NSF-CCF-1525372.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. "Shared Memory Consistency Models: A Tutorial." In *IEEE Computer*, Vol. 29 No. 12, December 1996.

[2] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. "Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming." *Technical Report HPL-2012-236, Hewlett-Packard*, 2012.

[3] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. "Atlas: Leveraging Locks for Non-volatile Memory Consistency." In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2014.

[4] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher M. Aycock, Gurushankar Rajamani, and David E. Lowell. "The Rio File Cache: Surviving Operating System Crashes." In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[5] Shimin Chen, Phillip B. Gibbons, and Suman Nath. "Rethinking Database Algorithms for Phase Change Memory." In *Conference on Innovative Data Systems Research*, 2011.

[6] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Optimistic Crash Consistency." In *Symposium on Operating Systems Principles*, 2013.

[7] Joel Coburn, Trevor Bunker, Meir Shwarz, Rajesh K. Gupta, and Steven Swanson. "From ARIES to MARS:Transaction

Support for Next-Generation Solid-State Drives." In *Symposium on Operating System Principles*, 2013.

[8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories." In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. "Better I/O Through Byte-Addressable, Persistent Memory." In *Symposium on Operating Systems Principles*, 2009.

[10] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. "Implementation Techniques for Main Memory Database Systems." In *International Conference on Management of Data*, 1984.

[11] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. "System Software for Persistent Memory." In *European Conference on Computer Systems*, 2014.

[12] Jim Gray and Andreas Reuter. "Transaction Processing: Concepts and Techniques." *Morgan Kaufmann Publishers, Inc.*, 1993.

[13] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. "NVRAM-aware Logging in Transaction Systems." In *Proceedings of the VLDB Endowment*, 2014.

[14] Intel. "Intel Architecture Instruction Set Extensions Programming Reference (319433-023)." `https://software.intel.com/sites/default/files/managed/0d/53/319433-023.pdf`, 2014.

[15] Intel and Micron. "Intel and Micron Produce Breakthrough Memory Technology." `http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-\technology`, 2015.

[16] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. "Aether: A Scalable Approach to Logging." In *Proceedings of the VLDB Endowment*, 2010.

[17] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. "Persistency Programming 101." In *Non-Volatile Memories Workshop*, 2015.

[18] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In *Communications of the ACM*, Vol. 21, July 1978.

[19] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. "Architecting Phase Change Memory as a Scalable Dram Alternative." In *International Symposium on Computer Architecture*, 2009.

[20] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. "Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory." In *Conference on File and Storage Technologies*, 2013.

[21] David E. Lowell and Peter M. Chen. "Free Transactions with Rio Vista.". In *Symposium on Operating Systems Principles*, 1997.

[22] Y. Lu, J. Shu, L. Sun, and O. Mutlu. "Loose-Ordering Consistency for Persistent Memory." In *International Conference on Computer Design*, 2014.

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation." In *Conference on Programming Language Design and Implementation*, 2005.

[24] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging." In *ACM Transactions on Database Systems*, Vol. 17 No. 1, 1992.

[25] Dushyanth Narayanan and Orion Hodson. "Whole-System Persistence." In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[26] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. "Telecom Application Transaction Processing Benchmark." `http://tatpbenchmark.sourceforge.net/`.

[27] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. "Memory Persistency." In *International Symposium on Computer Architecture*, 2014.

[28] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andreas C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "IRON File Systems." In *Symposium on Operating Systems Principles*, 2005.

[29] Alexander Thomson and Daniel J. Abadi. "The Case for Determinism in Database Systems." In *Proceedings of the VLDB Endowment*, 2010.

[30] Transaction Processing Performance Council (TPC). "TPC Benchmark B." `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf`.

[31] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. "Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory." In *Conference on File and Storage Technologies*, 2011.

[32] Tianzheng Wang and Ryan Johnson. "Scalable Logging through Emerging Non-Volatile Memory." In *Proceedings of the VLDB Endowment*, 2014.

[33] Haris Volos, Andres Jaan Tack, and Michael M. Swift. "Mnemosyne: Lightweight Persistent Memory." In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[34] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. "Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support." In *International Symposium on Microarchitecure*, 2013.

[35] Jishen Zhao, Onur Mutlu, and Yuan Xie. "FIRM: Fair and High-performance Memory Control for Peristent Memory Systems." In *International Symposium on Microarchitecure*, 2014.