ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory

Kai Wu
kwu42@ucmerced.edu

Jie Ren
jren6@ucmerced.edu

Ivy Peng†
peng8@llnl.gov

Dong Li
dli35@ucmerced.edu

University of California, Merced

Lawrence Livermore National Laboratory†

Abstract

Failure-atomic transactions are a critical mechanism for accessing and manipulating data on persistent memory (PM) with crash consistency. We identify that small random writes in metadata modifications and locality-oblivious memory allocation in traditional PM transaction systems mismatch PM architecture. We present ArchTM, a PM transaction system based on two design principles: avoiding small writes and encouraging sequential writes. ArchTM is a variant of copy-on-write (CoW) system to reduce write traffic to PM. Unlike conventional CoW schemes, ArchTM reduces metadata modifications through a scalable lookup table on DRAM. ArchTM introduces an annotation mechanism to ensure crash consistency and a locality-aware data path in memory allocation to increases coalesable writes inside PM devices. We evaluate ArchTM against four state-of-the-art transaction systems (one in PMDK [30], Romulus [21], DUDETM [46], and one from Oracle [50]). ArchTM outperforms the competitor systems by 58x, 5x, 3x and 7x on average, using micro-benchmarks and real-world workloads on real PM.

1 Introduction

Byte-addressable persistent memory (PM) can provide DRAM-like performance and storage-class capacity. The state-of-the-art Intel Optane DC PM could implement up to nine terabytes memory capacity on a single machine with latency in hundreds of *ns* [31, 32, 34, 57, 72]. Such high-performance PM is emerging in datacenters and clouds to boost performance-critical data-intensive applications, such as database [9,17,22,28,41,65] and graph workloads [18,25].

Crash consistency is a primary challenge in using PM. With PM, programs can recover their persistent data on PM even in the event of crashes. However, such a recovery requires a guarantee that persistent data is in a consistent state, a requirement referred as the crash consistency guarantee. Failure-atomic transactions are a popular mechanism to ensure crash consistency. Extensive studies [16,21,27,30,39,40,49–51,61,67,69,70,73] have proposed various transaction mechanisms that generally employ logging-based (undo or redo logging) or Copy-on-Write (CoW)-based designs.

Existing works optimize PM transactions by reducing data copying [11,20,51,68] or persistence overhead [20,35,38,43,56,62]. They emulate PM based on DRAM with increased memory latency or reduced bandwidth, but miss PM architecture details. In this study, we focus on the implications of PM architecture on transaction performance. Our performance analysis on state-of-the-art PM transaction systems identifies that the PM micro-architecture, such as internal buffers and data block size, has significant impacts on transaction performance. The mismatch between the transaction implementation and PM architecture can cause 3x-58x slowdown, compared to an architecture-aware implementation.

Performance characterization of PM architecture leads us to rethink the design of PM transactions. Logging-based transactions have a double write problem because of creating logs and updating data in-place. The excessive writes to PM mismatch with poor write performance on PM. CoW-based transactions avoid this problem, but suffers from performance overhead due to metadata updates, which causes many small writes misaligned with PM internal block size.

Therefore, high-performance PM transactions call for new design principles tailored to the characteristics of the emerging PM architecture, which is distinctive from conventional block devices and more than just a slower DRAM. We introduce two design principles customized to PM architecture.

- Avoid small (less than 256 bytes) writes to PM. Small writes in PM suffer from write amplification because data in a small write must be aligned with the internal write block size (256 bytes) in PM, which wastes memory bandwidth and delays transactions. Our characterization study reveals that in state-of-the-art PM transaction systems (one in PMDK [30], Romulus [21], DUDETM [46], and an Oracle transaction system [50]), more than 78% of data objects are smaller than 64 bytes, when the transaction systems perform write operations

on 512-byte persistent objects. The main source of those small data objects comes from metadata for transaction runtime state, memory allocation and object mapping.

- Encourage coalescable writes. Sequential write performs much faster than random write on PM (e.g., for 64-byte writes, sequential write is 3.7x faster than random write). Multiple sequential writes can be coalesced in an internal buffer of Optane, enabling high performance.

We follow the above principles in ArchTM. ArchTM uses a CoW-like design to avoid the double write problem in logging-based transactions. To avoid small writes, ArchTM stores metadata of memory allocator and data objects on DRAM to reduce frequent small random writes to PM. However, such a design suffers from a fundamental tradeoff between performance and crash consistency. In particular, metadata on DRAM, although leading to high transaction performance can be lost when a crash happens, leading to a problem of identifying crash consistency of data objects.

The above problem is caused by the fact that metadata is the only connection between the transaction state and data objects for crash recovery. Such a connection is not PM-oriented. Removing it causes isolation between transaction state and data objects. To address this challenge, ArchTM introduces a lightweight annotation mechanism. This mechanism adds data object metadata (object ID and size) and transaction ID into the data object, and adds transaction ID into the transaction metadata (i.e., the transaction state variable). The transaction ID is persistent and sets up an alternative connection between data objects and the transaction state. Using the transaction ID, the data object ID and size, ArchTM can easily locate data objects and identify their crash consistency after a crash.

To encourage coalescable writes, ArchTM makes best efforts to allow consecutive memory allocation requests to get contiguous memory allocations. This strategy is based on the observation that in a transaction, data objects that are allocated consecutively are likely to be updated together. For example, in a key-value store system, memory allocation requests for a key data object and a value data object associated with the key often happen together. Writes to the key and value data objects happen in sequential and continuous order. Hence, allocating the key and value contiguously in the address space likely results in coalescable write.

However, to implement the above strategy, we must re-examine the traditional wisdom for memory allocation. The existing memory allocators typically use multiple free lists for each thread. Each free list supports allocation requests for specific sizes. Such size-class-based memory allocation is used to reduce memory fragmentation. However, it allocates noncontiguous memory blocks to consecutive memory allocation requests if they are fulfilled by multiple free lists. Hence, there is a fundamental tradeoff between *allocation locality* and memory fragmentation.

To break this tradeoff and encourage coalescable writes, ArchTM uses a single free list and a lightweight online defragmentation mechanism. In particular, ArchTM supports locality-aware data path using the single free list for allocation and uses a recycle list to collect and merge freed memory blocks. For defragmentation, ArchTM aggregates data objects in highly fragmented memory regions to create large and contiguous memory blocks.

In summary, the paper makes the following contributions:

- We reveal the performance characterization of realistic PM hardware and pinpoint the performance problems in the representative PM transactions. Such problems are caused by the negligence of the characteristics of PM architecture in traditional PM transaction designs.
- We identify two fundamental tradeoffs to enable high performance PM transactions. We introduce a new PM transaction design, ArchTM, customized to the PM architecture and breaking the tradeoffs.
- ArchTM beats state-of-art PM transaction systems PMDK, Romulus, DUDETM and the Oracle system by 58x, 5x, 3x and 7x on average, using micro-benchmarks and real-world workloads on PM hardware.

2 Background

2.1 Persistent Memory Transactions

Failure-atomic transactions are a common solution to ensure crash consistency on PM [16, 21, 27, 30, 39, 40, 49–51, 61, 67, 69, 70, 73]. Updates in a failure-atomic transaction either all succeed or fail, leaving the data on PM in a consistent state. We refer to data objects accessed in a transaction as *persistent objects*. PM transactions are implemented in two major paradigms – logging and copy-on-write (CoW).

Logging-based transactions can use either *undo-logging* or *redo-logging*. Both logging approaches must write twice to update a persistent object, i.e., update the log and then the data (Figure 1 a and b). This in-place update to the data could cause concurrent random writes because transactional workloads could update arbitrary persistent objects.

CoW-based transactions create a new copy of a persistent object before modifying it (Figure 1 c). All updates are captured in the new copy, i.e., out-place updates. After persisting updates in the new copy, the system updates the pointer to the persistent object to the new copy and discards the old copy. Hence, CoW transactions write to PM only once. Even when random persistent objects are updated, persisting their new copies laid out sequentially still result in sequential writes.

2.2 Memory Management in PM Transactions

In logging or CoW paradigms, logs are inserted and removed, or copies of persistent objects are created and deleted in each transaction. Frequent memory allocation and deallocation in concurrent transactions require **scalable** solutions. Also, the

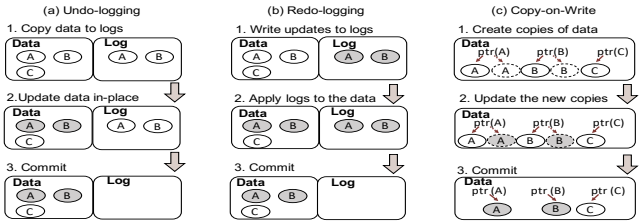


Figure 1: Three transaction implementations: undo-logging, redo-logging, and copy-on-write.

persistence in PM imposes unique requirements of **consistency** and **low fragmentation** on memory management.

Scalable memory allocators [14, 23, 26, 64], including state-of-the-art PM allocators [15, 30, 66], typically implement thread-local free lists and global free lists. An allocation request is first tried on the requester thread's local free list before being forwarded to the global free list. For a deallocation request, the freed memory block is added to the requestor thread's local free list to avoid synchronization on the global free list. The existing memory allocators usually predefine a set of object size classes. For each size class, the allocator maintains a list of free memory blocks of that size. An allocation request is fulfilled by the list in the nearest size class. Memory fragmentation occurs when the selected size class is larger than the requested size. Unlike volatile memory, fragmentation on PM has a longer-lasting impact. Volatile memory may restart the program to diminish fragmentation while fragmentation on PM persists through restarts. Besides, a PM allocator needs to ensure its metadata in a consistent state to avoid data loss and memory leakage after crash.

2.3 Emerging PM Architecture

Emerging persistent memories are byte-addressable, and PM DIMMs are attached directly to the memory bus, like conventional DRAM DIMMs. Processors can access PM through `load` and `store` instructions. The Intel Optane DC PM represents state-of-the-art PM hardware [34, 57, 58, 71]. The data transfer between the processor and PM occurs at the cache line granularity (64 bytes). The Optane internal transactions, however, have a granularity of 256 bytes. *Write amplification* occurs as a result of the two mismatched transaction sizes. For instance, updating a cache line (64 bytes) could result in a 256-byte write inside the Optane media. A *combining buffer* of 16KB [34] sits inside each NVDIMM to coalesce writes. Multiple writes from the processor could be combined into a single transaction if they occupy a contiguous 256-byte block.

3 Performance Characterization

We study the performance of PM transactions and Optane PM to gain insights for our design.

3.1 Transaction Performance Study

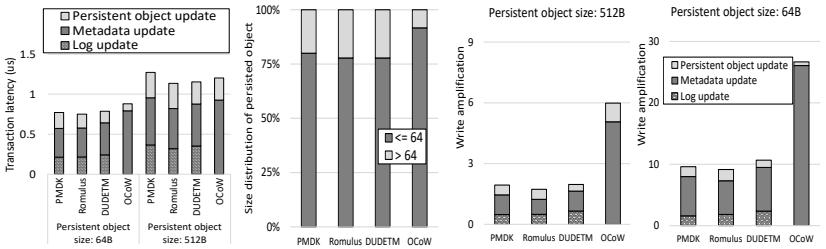
We study four representative PM transaction systems: PMDK [30], Romulus [21], DUDETM [46], and one from Oracle [50]. PMDK uses undo-logging, Romulus and DUDETM use redo-logging, and the Oracle system (denoted as *OCoW*) uses CoW. The specification of our Optane platform is in Section 6. We focus on write operations because they are the most expensive transaction operation, and writes to PM are expensive. A write operation in a transaction needs to update persistent object, log (if logging-based), and metadata. Figure 2a shows the latency breakdown of a write operation in PM transactions. We report the performance on small (64-byte) and large (512-byte) persistent objects. The figure shows that most time is spent on log updates or metadata updates.

We instrument the APIs used to persist data objects (e.g., *pmemobj_persist*() in PMDK) to study the performance of write operations. The APIs use the starting address and size of the data objects as input. Figure 2b reports the distribution of the *persisted* data size in transactions that perform write operations on 512-byte persistent objects. The figure reveals that more than 78% of persisted objects are smaller than 64 bytes, i.e., a lot of small writes on PM. Furthermore, we study write amplification, quantified as the ratio between write traffic in PM measured by performance counters and the number of bytes modified by transactions. Figure 2c reports the write amplification in transactions that perform write operations on 64- and 512-byte persistent objects. All systems exhibit write amplification, inflating PM write traffic by 1.8x - 27x.

Performance analysis. *We find that the metadata updates are the primary source of small writes.* In general, transaction systems have four types of metadata: metadata for transaction runtime, metadata for memory allocation, log metadata, and metadata for persistent objects. Metadata for transaction runtime records transaction status, e.g., COMMIT or ABORT, and transaction IDs. Metadata for memory allocation has information about memory consumption. Log metadata has information on logs (e.g., the indexing of log records), and is unique in logging-based transactions. Metadata for persistent objects store pointers to the new or old copy of persistent objects, and is unique in CoW-based transactions. By design, CoW-based systems have more metadata updates than logging-based ones. For instance, OCoW has about 270% more metadata updates than the other three logging-based systems. For each update, a CoW-based transaction must allocate a new data copy, remap pointers to the data, and deallocate the old data copy. This process generates frequent small writes to metadata for memory allocation and persistent objects.

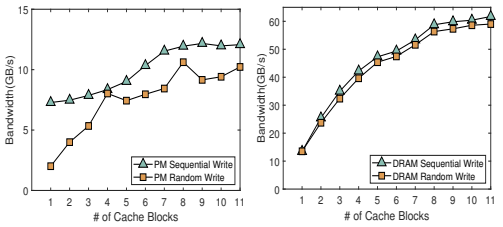
3.2 Performance Study of PM Writes

We study the write performance on Optane DC PM using a microbenchmark that performs random and sequential writes. Each write is followed by cache line flushes to persist to PM.



(a) Latency breakdown **(b)** Size distribution **(c)** Write amplification

Figure 2: Performance characterization of write operations in PM transactions.



(a) PM **(b)** DRAM

Figure 3: Sequential and random write bandwidth at different write sizes on PM and DRAM.

Various write sizes, ranging from one to 11 cache lines, are tested. Figure 3 reports the bandwidth of performing 100M writes using 24 threads on PM and DRAM. We have the following observations and insights for high-performance PM transactions.

Figures 3a and 3b show that write bandwidth of PM is significantly lower than that of DRAM. On our system, write bandwidth to DRAM reaches 80 GB/s but only 13 GB/s to Optane PM. Furthermore, on Optane PM, the peak write bandwidth is 13 GB/s, 3x lower than the peak read bandwidth. These results are consistent with the existing work [34]. Hence, **reducing write traffic on PM is critical** for high-performance transactions. The logging-based transaction systems need to write data twice to update a persistent object, which causes excessive write traffic.

Figure 3a shows that small random writes on PM perform worse than sequential writes. When writing only 64 bytes (Figure 3a), random write merely achieves 25% of the bandwidth of sequential write. This performance gap is caused by the 256-byte Optane internal granularity and write amplification, and the gap reduces when the write size increases. The logging-based transactions update persistent objects in-place. This could result in random writes, because persistent objects in a transaction can be randomly distributed on PM. Using out-place updates, as in CoW-based transactions, can enable sequential writes because the new copies of persistent objects are manageable and can be laid out contiguously in PM.

Figure 3a shows that the random writes on PM have performance spikes at write sizes that are a multiple of 256 bytes, e.g., four and eight cache lines. In contrast, random writes on DRAM (Figure 3b) exhibits no such pattern. Such performance on PM is due to the effect of the write combining buffer. It buffers and combines 64-bytes stores into a 256-byte internal store. Small simultaneous writes to contiguous address space are more likely to be combined into one internal store than small writes to arbitrary addresses. Therefore, increasing the probability of concurrent writes to contiguous address space can increase the opportunity to **leverage the combining buffer** hardware to coalesce writes inside the PM.

4 Design Principles and Major Techniques

Driven by the performance characterization and analysis of existing PM transactions and PM, we introduce two design principles and five techniques in ArchTM for high-performance architecture-aware transactions.

- **Avoid small writes on PM.**
  (1) Logless. ArchTM favors the CoW mechanism to reduce write traffic to PM.
  (2) Minimize metadata modifications on PM with guaranteed crash consistency. ArchTM keeps transient metadata on DRAM to avoid frequent metadata modifications on PM. Also, ArchTM introduces an annotation mechanism to connect the persistent transaction state with data objects. From the transaction state of data objects, ArchTM can detect the consistency of data on PM and recover from a crash.
  (3) Scalable persistent object referencing. ArchTM uses a scalable object lookup table on DRAM to quickly locate the latest copies of persistent objects in concurrent transactions.
- **Encourage coalescable writes**.
  (4) Consecutive allocation requests get contiguous memory blocks. ArchTM supports a locality-aware data path for small memory allocations to encourage sequential writes in transactions.
  (5) Avoid memory fragmentation. ArchTM employs a lightweight online memory defragmentation technique that examines memory usage by regions and reduces fragmentation on PM.

4.1 Logless

ArchTM employs a CoW-like mechanism to reduce write traffic to PM. Upon an update request, ArchTM creates a new copy of the persistent object and applies updates to the new copy. The out-of-place update in CoW reduces the number of PM writes. When committing the new copy to PM, consecutive writes into contiguous memory addresses increase the possibility of writes coalesced at the combining buffer. However, naively adopting CoW incurs excessive metadata

updates on PM due to object remapping and allocation management (Section 3.1). We address this challenge by maintaining metadata on DRAM.

4.2 Minimize Metadata Modification on PM

ArchTM places the memory allocation metadata on DRAM. It does not record memory allocation and reclamation into logs on PM as in previous PM transaction systems [19, 21, 30, 66, 69]. Also, ArchTM avoids modifying the persistent object metadata on PM by using an *object lookup table* on DRAM. This lookup table is used to locate the latest copy of a persistent object quickly. Existing CoW-based implementations [50] must modify the persistent object metadata on PM to update the pointer to the object to the new copy (Figure 1.c). With these metadata in DRAM, ArchTM reduces small PM writes and accelerates the lookup, but cannot ensure crash consistency. ArchTM introduces an annotation mechanism to guarantee crash consistency.

Annotation. ArchTM annotates a transaction by adding a transaction ID into the transaction metadata (the transaction state variable). The embedded transaction ID is persisted immediately when the transaction state changes to *start*. ArchTM also annotates a persistent object by adding the object information, i.e., object ID, object size, and transaction ID, into the object header on PM when the object is created. During the recovery from a crash, ArchTM uses the object ID and size to identify each persistent object on PM. Then, ArchTM uses the annotated transaction ID to identify the most recent copy of a persistent object, recycle the stale copies, and discard uncommitted modifications.

4.3 Scalable Object Referencing

ArchTM uses an object lookup table to find the critical information, such as the location of the latest copy of a persistent object. The table is indexed by persistent object IDs. When a persistent object is allocated, the allocator thread gets an object ID and populates the corresponding entry in the lookup table. Multiple threads can reference persistent objects from the table concurrently and efficiently because DRAM supports higher bandwidth than PM.

The object lookup table is essential for high-performance transactions. Compared to decentralized object referencing [40, 50], the object lookup table in ArchTM resides on a contiguous DRAM space, which brings convenience for management (e.g., checkpointing) and migration. If the DRAM space is insufficient to store the whole lookup table, the spilling part of the table is placed on PM. Compared with general concurrent index data structures, such as hash tables, our object lookup table is easy to implement and has no synchronization overhead. The competition between threads to get an entry from the lookup table cannot happen, because threads are assigned with disjoint sets of object IDs and hence update disjoint sets of table entries. The object lookup table can find the object metadata in one step because it uses the object ID as the index of the table, which differs from other indexes (e.g., hash table and B-trees) that require additional calculations or queries to find object metadata.

4.4 Contiguous Memory Allocations

ArchTM customizes memory allocation and reclamation for transactional workloads on PM to maximize the possibility of sequential writes. Small allocations are the main optimization focus because sequential writes benefit small objects more than large objects (See Figure 3a). In ArchTM, there are two data paths for persistent object allocation and reclamation: (1) a regular data path for large allocations and reclamations, similar to existing allocators like JEMalloc [23]; and (2) a locality-aware data path for small allocations. The latter optimizes through a single free list and global recycling procedure.

A single free list is used in ArchTM for allocating objects of various sizes. Existing approaches [14, 15, 23, 26, 30, 64, 66] use multiple free lists, each for a different allocation size. Multiple free lists could cause consecutive allocation requests of different sizes to go to different free lists. Consequently, those requests get noncontiguous memory allocations, and writing to them leads to nonsequential writes to PM. Instead, using a single list of freed segments in sorted order would encourage consecutive requests to get sequential allocations. To maximize concurrency, ArchTM assigns each thread with a dedicated portion from the global free list (Section 5.1).

Recycle and merge memory blocks globally. Current approaches [14, 15, 23, 30, 64, 66] return freed memory blocks to thread-local free lists directly. This procedure avoids synchronization on managing a global free list but may harm the locality of freed memory blocks. Free memory blocks in a free list may be noncontiguous so that consecutive allocation requests get noncontiguous allocations. ArchTM runs a helper thread to collect and merge freed blocks from threads. These freed blocks are sorted and merged into a global recycle list before returning them to the global free list. The global recycling procedure does not happen in the critical path and does not affect the efficiency of memory deallocation.

4.5 Reduce Memory Fragmentation

Using a single free list for various allocation sizes could result in memory fragmentation. ArchTM uses a 64-byte size class in the memory allocator. An allocation smaller than the size class gets rounded up. We choose this size class to avoid false sharing in cache lines.

ArchTM introduces an online defragmentation mechanism to reduce memory fragmentation. The mechanism monitors the memory usage of the persistent object pool in the background to identify underutilized memory regions. During the memory allocation, this mechanism dynamically aggregates

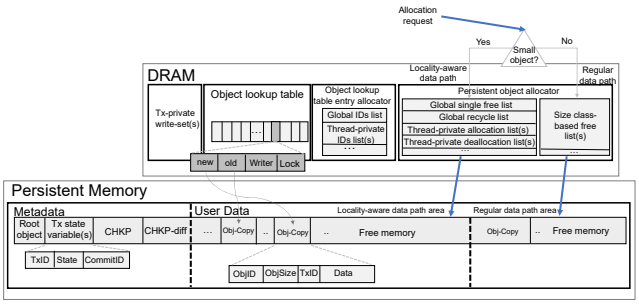


Figure 4: Major data structures in ArchTM

persistent objects distributed in the underutilized memory regions to improve memory usage. The online defragmentation mechanism is a user-space solution that can be enabled or disabled. It requires no modifications to operating systems as required by existing solutions [52]. Also, the user-space solution is more flexible than offline static solutions [59] and can react to changes in the application during execution.

5 ArchTM Implementation

We describe our implementation based on Section 4.

5.1 Data Structures

Persistent Data Structures on PM. ArchTM maintains a persistent memory pool partitioned into metadata and user data areas. As depicted in Figure 4, the metadata area stores a root object, a list of transaction state variables, a checkpoint field (*CHKP*), and a checkpoint-diff field (*CHKP-diff*).

The list of transaction state variables records the state of each ongoing transaction. Each variable encodes transaction state and ID, and commit ID. We use the transaction start timestamp as transaction ID, and the transaction commit timestamp as commit ID. They are global timestamps captured at the beginning and end of a transaction. ArchTM uses hardware clock (*rdtscp* in x86 architectures [6, 36]) and prevents the constant skew of the hardware clock among processors by the ORDO primitive [36] to ensure correct ordering of transactions. The transaction state indicates the progress of a transaction, e.g., BEGIN, COMMITTED, END or ABORT.

CHKP stores a persistent checkpoint of the object lookup table to speedup recovery (Section 5.6). *CHKP-diff* records the list of memory blocks (named *memory segments*) preallocated to each thread (Section 5.4-Allocation). *CHKP-diff* is useful to track working objects before the next checkpoint. It is implemented as an array of elements containing three fields: ID of ongoing transactions for which the segment is fetched, the segment start address and size.

The user data area stores persistent objects. Each object has an object header and data. The header contains object ID and size, and transaction ID. The user data area is divided into a regular data path area for large object allocations and a locality-aware data path area for small object allocations.

Transient Data Structures on DRAM ArchTM maintains an object lookup table and a hash set per transaction. The object lookup table is a one-dimensional array mapping a persistent object ID to a persistent object on PM. Each PM object has an entry in the table. An entry has four fields, i.e., a pointer to the latest copy (*new*), a pointer to the old copy (*old*), a variable (named *writer*) storing the pointer of the transaction state variable of the ongoing transaction that modifies the latest copy, and a write lock associated with the *writer* to coordinate parallel transactions. The hash set (named *write-set*) is used to collect the IDs of all persistent objects modified by a thread in an active transaction. Before committing a transaction, all objects in the hash set must be persisted.

ArchTM manages metadata for two allocators on DRAM. The first allocator allocates an entry in the object lookup table when a persistent object is created. This allocator maintains a list of free IDs for persistent objects (named *ID list*) per core. A persistent object ID is the index of an entry in the object lookup table. When the allocator allocates an entry, it gets an object ID from the ID list. When a persistent object is freed, its object ID is returned to the ID list. We reuse IDs for persistent objects to avoid the explosion of IDs. New IDs are created only when the ID list is empty.

The second allocator allocates persistent objects. It reuses the metadata structures in JEMalloc [23] for the regular data path but adds significant extensions to optimize small writes to PM (Section 4). For the locality-aware data path, ArchTM maintains a global free list and a global recycle list. The global free list contains memory blocks available for allocations. To ensure sequentially when multiple threads access the global free list, ArchTM uses a write lock on the global free list. To mitigate contention on the global free list, each thread maintains a thread-private allocation list, which is a portion from the global free list. Only when a thread exhausts its allocation list will the thread access the global free list to get a new portion. Therefore, synchronization on the global free list is infrequent. The global recycle list collects memory blocks freed by all threads. The allocator manages a deallocation list per thread to collect deallocated memory blocks. Blocks from these thread-local deallocation lists are gathered, sorted, and merged into the global recycle list. Memory management is described in detail in Section 5.4.

5.2 Background Threads

Background threads are helper threads transparent to the application. ArchTM uses two background threads to manage the PM pool at runtime – the *garbage collection (GC) manager* and the *fragmentation manager*. The GC manager recycles freed persistent objects. The fragmentation manager examines memory usage by regions and aggregates memory blocks for defragmentation (see Section 5.4).

Algorithm 1 Start, read, and write operations.

```
1:  function APT_TX_BEGIN
2:      volatile TxID = GLOBALTIMESTAMP()
3:      TxState.ATOMIC_STORE(TxID, BEGIN)
4:      Fence()
5:  end function
6:
7:  function APT_TX_READ(TxState, objID)
8:      obj = objLookupTable[objID]
9:      if obj.new == NULL then return obj.old
10:     end if
11:     if obj.writer → TxID == TxState.TxID then return obj.new
12:     end if
13:     if obj.writer → State == COMMITTED and obj.writer → CommitID <=
        TxState.TxID then return obj.new
14:     end if
15:     return obj.old
16: end function
17:
18: function APT_TX_WRITE(TxState, objID)
19:     obj ← objLookupTable[objID]
20:     if obj.new! = NULL and obj.writer → TxID == TxState.TxID then
21:         return obj.new
22:     end if
23:     if LOCK(obj.writer) then
24:         obj.writer = &TxState
25:         obj.new = ALLOC(obj.old)
26:     else ABORT_AND_RETRY( )
27:     end if
28:     obj.new = DUPLICATE(obj.old)
29:     obj.new.header.txID = TxState.TxID
30:     # append the object to write-set
31:     write_set.insert(objID)
32:     return obj.new
33: end function
```

Algorithm 2 Commit and post-commit operations.

```
1:  function APT_TX_ON_COMMIT(TxState)
2:      if EMPTY(write_set) then return
3:      end if  # read-only tx
4:      for each obj ∈ write_set do FLUSH(obj.new)
5:      end for  Fence()  # persist all modified objects
6:      volatile CommitID = GLOBALTIMESTAMP()
7:      TxState.ATOMIC_STORE(COMMITTED, CommitID)
8:      Fence()
9:      APT_TX_POST_COMMIT(TxState)
10: end function
11:
12: function APT_TX_POST_COMMIT(TxState)
13:     for each tx ∈ Ongoing_TXs do
14:         if tx.TxID < TxState.CommitID then WAIT_FOR(tx)
15:         end if
16:     end for
17:     while obj ← write_set.pop() do
18:         FREE(obj.old)  # append to the reclaim list
19:         obj.old = obj.new
20:         obj.new = NULL
21:         obj.writer = NULL
22:         UNLOCK(obj.writer)
23:     end while
24:     TxState.ATOMIC_STORE(END, INF)
25:     Fence()
26: end function
```

5.3 Transaction Operations

ArchTM supports five core operations to begin, read, write, commit, and postcommit in a transaction. ArchTM provides snapshot isolation [8, 13] similar to existing work [12, 27, 45, 48, 55, 63] and industrial production database systems [2–5, 7, 53]. We illustrate the operations in Algorithms 1 and 2.

APT_TX_BEGIN starts a transaction and assigns a unique ID (*TxID*) based on the global timestamp (Alg. 1 Line 2) to the transaction. A transaction state variable (*TxState*) is created and stored in the metadata area on PM. *TxState* is a combination of the *TxID*, state and transaction commit ID (*CommitID*). At the transaction beginning, ArchTM adds *TxID* and the state BEGIN into *TxState* by an atomic write.

APT_TX_READ returns a pointer to a copy of the persistent object with (*objID*). If the object is not being updated by any transactions (Alg. 1 Line 9), the pointer to the old copy is returned. If the object is being updated by the current transaction (Alg. 1 Line 11) or a transaction committed before the current transaction starts (Alg. 1 Line 13), the pointer to the new copy is returned. Otherwise, ArchTM returns the pointer of the old copy. The whole process is lock-free.

APT_TX_WRITE returns a pointer to the persistent object *objID* ready for update. If the persistent object already has a new copy and the most recent update to the copy is performed by the current transaction, the pointer to the new copy is returned (Alg. 1 Lines 20-22). If the persistent object does not have a new copy, the application thread allocates a one, acquires the write lock of the *writer* of the object (Alg. 1 Line 23), duplicates the old copy to the new one, and then updates the new copy. The application thread also inserts the object ID into the *write-set*. If the application thread fails to obtain the write lock of the object, *APT_TX_WRITE* aborts and retries in a new transaction.

APT_TX_ON_COMMIT commits a transaction. If the transaction is read-only, no persistent operations are performed. Otherwise, ArchTM persists the modified objects recorded in the *write-set* to PM (Alg. 2 Lines 4-5). After that, ArchTM gets a global timestamp as CommitID and updates the state to COMMITTED with CommitID in the transaction state variable by an atomic write.

APT_TX_POST_COMMIT cleans up a committed transaction. First, it checks whether there is any ongoing transaction that starts before the current transaction is fully committed (Alg. 2 Lines 13-16). It reclaims the old copy (i.e., putting the old copy in the thread-private deallocation list) after the earlier transactions are fully committed. This ensures that the old copy of the persistent object is no longer required in any ongoing transaction. Afterwards, ArchTM sets the new copy as the old copy and sets the new copy as *NULL*. Finally, it resets and unlocks the writer of modified objects. ArchTM also updates and persists the transaction state to END and CommitID to *INF*.

5.4 Memory Management for Transactions

ArchTM uses a customized persistent object allocator. Depending on the size of an allocation request, ArchTM chooses the locality-aware data path for small allocations and use the regular data path for the others. We describe the locality-aware data path in this Section.

Allocation. When a thread attempts to allocate a persistent object, ArchTM searches through the thread's private allocation list to locate the first memory block larger than the requested size. If no block is found, ArchTM fetches freed memory blocks from the global free list to refill the allocation list. Each fetch takes a large and fixed-size memory segment to avoid frequent contention on the global free list. The fetching history is stored and persisted in *CHKP-diff*. Each fetching event in *CHKP-diff* contains the IDs of ongoing transactions, where the segment is fetched from, the segment start address, and the segment size. If ArchTM cannot find free memory blocks from the global free list, ArchTM replenishes memory blocks from the global recycle list to the global free list.

Deallocation (garbage collection). When a thread deallocates a persistent object, the object is ready for GC because no other transactions are accessing the object (Alg. 2 Lines 13-16). The deallocated object is added to the thread's private deallocation list. In the background, the GC manager periodically collects freed objects from threads to the global recycle list, during which freed blocks are zeroed. Synchronization between application threads and the GC manager is rare because an application thread only updates the head while the GC manager only updates the tail of a deallocation list. The global recycle list is sorted to speed up search during allocation and fragmentation ratio computation during defragmentation. Sorting is inexpensive because when freed memory blocks are added to the global recycle list, they are already mostly sorted.

Defragmentation. ArchTM implements an online defragmentation mechanism to improve the memory usage of the global recycle list. The mechanism works at the granularity of memory regions (4KB). The defragmentation manager monitors the fragmentation ratio (defined as the ratio of used memory to 4KB) of each memory region in the global recycle list. A memory region with a fragmentation ratio greater than f (f is 50% in our evaluation) is deemed underutilized. ArchTM aggregates persistent objects in underutilized regions and migrates them to a newly allocated memory region. For migration, the defragmentation manager internally creates a "mock" write transaction to ensure the atomicity of data migration and correctness. At the end of the "mock" write transaction, the migrated objects in the original location will be reclaimed through the deallocation process.

5.5 Recovery Management

ArchTM follows a two-step recovery process to resume the program from a crash.

1) Detect uncommitted transactions: This is implemented by checking the state of each transaction state variable on PM. If a state is neither COMMITTED nor END, ArchTM inserts the transaction ID of the uncommitted transaction into a temporary buffer (named $uncommittedTxIDs$).

2) Rebuild object lookup table: ArchTM creates a new object lookup table on DRAM (described in Section 5.1) and loads the object information to the new table. The loading process is similar to processing write operations, with the difference that the object information is retrieved from PM instead of the user request. In particular, ArchTM scans the user data area on PM to find persistent objects and inserts their location information (i.e., pointers to the objects on PM) into the lookup table. ArchTM puts the location information of each persistent object in the lookup table based on the object ID which indicates where the location information is in the original lookup table. To identify an object on PM, ArchTM relies on the object header annotated in each persistent object. The header contains the object ID and object size, which is used to isolate persistent objects from each other on PM.

ArchTM must eliminate object copies in uncommitted transactions. If the transaction ID of an object copy is found in $uncommittedTxIDs$, the object copy is discarded, and its memory space is reclaimed.

Since ArchTM does not invalidate the memory blocks of a freed object copy until the memory manager recycles them to the global recycle list, a persistent object may have multiple copies in the PM pool. Therefore, ArchTM must identify the latest copy and discards the others. When ArchTM reads a persistent object from PM and finds that the object already exists in the object lookup table, ArchTM compares the transaction IDs annotated in these two copies and only keeps the latest one. The mapping information in the object lookup table is then updated, and the old copy is reclaimed.

Crash consistency is ensured because (1) all modifications in uncommitted transactions are discarded, (2) all modifications in a committed transaction are persisted, and (3) only the latest committed copy of a persistent object is retained. All uncommitted transactions are captured in the transaction state variables stored in PM, and all object copies with a transaction ID in these uncommitted transactions are discarded during recovery. A transaction is only marked committed after all modified persistent objects in this transaction (collected in *write-set*, (Alg. 1 Line 31)) are persisted (Alg. 2 Lines 4-5). ArchTM identifies the latest committed copy of an object by transaction IDs, which by design guarantees that a transaction ID is no earlier than the commit ID of another transaction if they update the same object (Alg. 1 Lines 23-27).

5.6 Reduction of Recovery Time

The recovery process may take a long time if a large number of persistent objects exist on PM because ArchTM must scan the entire user data area to locate objects and rebuild the object lookup table. The recovery can take as long as tens of minutes on PM with TBs of capacity.

We reduce the recovery time by incorporating an incremental checkpoint technique into ArchTM. In particular, ArchTM periodically copies the modifications of the object lookup table since the last checkpoint to PM, such that ArchTM builds

a checkpoint of the object lookup table on PM. When restarting from a crash, ArchTM uses the checkpoint to resume the object lookup table, instead of building it from scratch.

ArchTM uses the following method to detect modifications of the lookup table since the last checkpoint. After taking an incremental checkpoint, ArchTM temporarily blocks all transactions, sets all pages of the object lookup table on DRAM as read-only by enabling write protection, and then resumes the transactions. Any following writes to those pages will trigger a write-protection page fault, indicating that the page is modified. ArchTM records the faulted pages for the next incremental checkpoint. After a page fault is triggered, the page is not write-protected, and there will be no more page faults. At the time of incremental checkpoint, only those modified pages are copied from DRAM to PM.

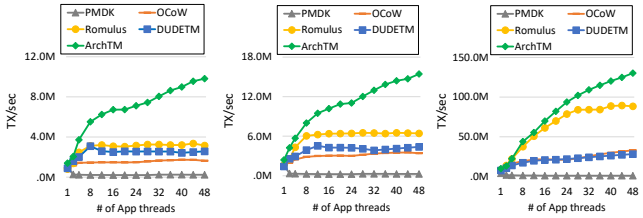
Using a persistent checkpoint of the object lookup table for recovery is not enough to reduce recovery time, because after a crash, the updates on object metadata since the last checkpoint are lost. To solve this problem, the persistent object allocator in ArchTM records the fetching history of memory segments in *CHCP-diff* (Section 5.4-Allocation), and those PM segments contain the modifications of persistent objects since the last checkpoint. ArchTM scans those modified segments to find missing updates as Section 5.5. Note that page information collected from the above page fault mechanism cannot be used to locate missing segments, because it is on DRAM and gets lost after crash. The page information is only used to implement incremental checkpoint. Overall, ArchTM uses a combination of the checkpoint of the object lookup table and the fetching history of memory segments in *CHCP-diff* to quickly restore the object lookup table.

6 Evaluation

We use an Intel Purley platform that has 2nd Gen Intel® Xeon® Scalable processor, 32KB L1 caches, 1MB L2 caches, and a shared 35MB L3 cache. The memory subsystem consists of 12 DRAM DIMMs and PM DIMMs, providing a total of 192 GB DRAM and 1.5 TB PM. We compare ArchTM with four state-of-the-art transaction systems: PMDK [30], Romulus [21], DudeTM [46], and OCoW [50]. PMDK uses libpmemobj v1.7. Libpmemobj does not support isolation, so we use a readers-writer lock to protect a transaction from concurrent accesses. Romulus uses RomulusLR for the best performance, and DudeTM uses the default persistent scheduler. We set the checkpoint frequency in ArchTM to 30 seconds, and the size of the pre-allocated PM segment to two GB. The granularity of memory regions for defragmentation (Section 5.4) is 4KB.

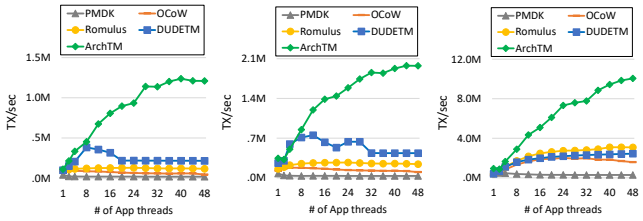
6.1 Micro-benchmarks

Hash tables and red-black trees are two important concurrent data structures widely used in database workloads [24, 37, 44, 60]. We evaluate hash tables and red-black trees with three



(a) 80% update **(b)** 50% update **(c)** 5% update

Figure 5: Performance and scalability of hash table.



(a) 80% update **(b)** 50% update **(c)** 5% update

Figure 6: Performance and scalability of red-black trees.

update rates (5%, 50%, and 80%) similar to [21, 27, 40, 61, 74]. Each transaction operation randomly accesses a key-value pair to read or update. Each key-value pair uses an 8-byte key and 16-byte value. Figure 5 and 6 present the performance and scalability results.

Hash table. The experiments use a hash table of 10K buckets, each as a single linked list. The hash table is initialized with 100K key-value pairs. ArchTM outperforms the other systems by 10x, 12x and 22x on average at 80%, 50%, and 5% update rates respectively (Figure 5). ArchTM demonstrates high scalability as the concurrency in applications increases to the maximum. In contrast, Romulus stops scaling, and DUDETM and OCoW have performance degradation when the application uses more than 16 threads.

In write-intensive workloads (Figures 5a and 5b), the sequential write technique contributes significant improvement at low application concurrency. When the number of application threads continues increasing, contention on the Optane media outweighs the write amplification. Other optimizations in ArchTM, such as the transient metadata on DRAM, start coping with this new bottleneck, and sustain performance scaling. In a read-intensive workload (Figure 5c), ArchTM achieves nearly linear speedup through scalable object referencing on DRAM and lock-free read operations.

Romulus scales well when the concurrency is low (i.e., 1-8 threads) for write-intensive workloads. At high concurrency, its single-threaded write operations become a performance bottleneck. DUDETM cannot consume volatile logs from DRAM to PM in time, causing long delays. OCoW has frequent metadata updates on PM for object remapping, allocation, reclamation, thereby reducing the overall throughput. PMDK shows the worst performance because it uses read-write locks extensively for logging and memory allocation.

Red-black tree. In this experiment, the red-black trees are initialized with one million key-value pairs. ArchTM outperforms the other systems by 7x-13x on average. It exhibits near-linear scalability as the number of threads increases for the read-intensive workloads (Figure 6c).

We notice that all three workloads have performance fluctuation at about 28 application threads, likely caused by the high contention on the Optane media. This contention point arrives later than that in the hash table, because each update in the red-black tree needs to search longer than in the hash table, reducing its write intensity.

PMDK, OCoW, Romulus, and DUDETM have lower scalability in the red-black tree than in the hash table. In write-intensive workloads (Figures 6a and 6b), the performance in these systems either fails to scale or even degrade when the concurrency increases. They suffer from the expensive synchronization [27, 40]. The lock-free operations and scalable object referencing in ArchTM avoid this contention and enables high performance at high concurrency.

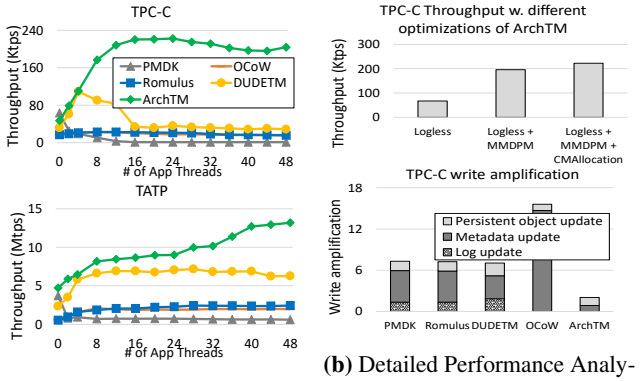
6.2 Real World Workloads

We run TPC-C [42] and TATP [54]) against PMEMKV [1]. PMEMKV is a in-memory key-value store developed by Intel. In this experiment, we use its *cmap* storage engine.

TPC-C. We run the *new-order* transaction test, where each application thread works on its corresponding warehouse and executes new order transactions. This workload has a 100% update rate. On average, each transaction inserts more than ten new objects into different tables and modifies more than ten existing objects. ArchTM significantly outperforms others by 10x, 9x, and 5x on average (Figure 7a). PMDK is more than 100 times slower than others when more than 12 threads are used. The performance of ArchTM scales up quickly to 24 application threads and then slightly declines due to write contention on the Optane media. DUDETM only scales up to eight threads because its performance is limited by centralized persistent logs. Once the background thread cannot flush the log buffer to PM in time, the application threads are delayed.

TATP. TATP is widely used for online transaction processing. ArchTM outperforms DUDETM, Romulus, OCoW and PMDK by 2x, 6x, 5x, and 13x, respectively. For evaluation, we implement three read-only and three read-write transactions similar to [27, 46]. The transactions in TATP are less write-intensive than the TPC-C test. Therefore, ArchTM achieves performance scaling up to the maximum application threads. Since TATP has less write traffic than TPC-C, DUDETM sustains performance at 16 threads and beyond.

We quantify the contribution from our design techniques to performance improvement. We separate techniques into logless, minimized metadata modification on PM (*MMDPM*), and contiguous memory allocation (CMAlloction). Figure 7b compares the performance using different techniques when running TPC-C with 24 application threads. In this test, We



(a) Throughput

(b) Detailed Performance Analysis on TPC-C

Figure 7: Real-world workloads with PMEMKV.

use DUDETM as the baseline, and its throughput is 37 Ktps. Minimized metadata modification on PM contributes the most (66%) performance improvement. The logless design and the contiguous memory allocation technique contribute 18% and 16% performance improvement, respectively. Using the same test configuration (Figure 7b-bottom), we quantify the write amplification in the five systems. The write amplification in ArchTM is only 2.03. ArchTM has 3x to 8x lower write amplification than the other systems.

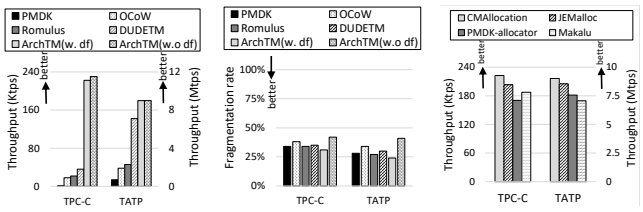
6.3 Performance Analysis

Online defragmentation. We evaluate the online defragmentation technique by quantifying the system throughput and memory fragmentation rate in TPC-C and TATP against PMEMKV. Each test uses 24 application threads. We compare the performance of ArchTM with and without online defragmentation (denoted as w.df and w.o.df in Figure 8), with four other PM systems.

The two ArchTM-based systems outperform other systems by 12x and 3x on average on TPC-C and TATP, respectively. The online defragmentation in ArchTM reduces memory fragmentation from 58% to 69% with only 3% overhead on system throughput on TPC-C. TATP is less write-intensive than TPC-C, and therefore no noticeable performance loss is observed from the online defragmentation. Figure 8b reports the memory fragmentation rate of all systems. The memory fragmentation rate of the ArchTM with online defragmentation is 4%, 9%, 3%, and 5% lower than PMDK, OCOW, Romulus, and DUDETM respectively. *ArchTM with online defragmentation is 14% lower that without it, demonstrating the necessity of using our online defragmentation.*

Contiguous memory Allocation. We evaluate the effectiveness of contiguous memory allocation (*CMAllocation*) in ArchTM. For comparison, we port ArchTM to use three state-of-the-art allocators, i.e., JEMalloc [23], PM allocator in PMDK [30], and Makalu [15]). Figure 8c reports the system throughput when ArchTM is equipped with the different allocators in TPC-C and TATP against PMEMKV.

The CMAllocation-based system achieves 9% and 6%



(a) Impact of online defragmenation. **(b)** Memory fragmentation. **(c)** Improvment from CMAlloction.

Figure 8: Evaluate the effectiveness of online defragmenation and contiguous memory allocation.

higher throughput than JEMalloc-based system on TPC-C and TATP, respectively. It also offers 20% and 18% higher throughout than PMDK- and Maruku-based systems. The customized locality-aware data path enables CMAllocation to encourage sequential writes on PM for better performance. In the PMDK and Maruku allocators, the poor scalability and frequent metadata updates become the bottleneck.

Checkpoint and Recovery Time. The checkpoint frequency trades off system throughput with recovery time. We vary the frequency from one second to 60 seconds in TPC-C against PMEMKV. We compare the system throughput with and without checkpoints, and find that checkpoints impose 11% overhead at the highest checkpoint frequency (i.e., one second). At a moderate checkpoint frequency, e.g., 30 seconds, the throughput loss diminishes to less than 1%.

We trigger a random crash after the program runs two minutes and then time the recovery. As expected, the recovery time increases linearly as the checkpoint frequency decreases. For the 30 GB workload set of TPC-C, ArchTM recovers the system in eight seconds at a checkpoint interval of 30 seconds and the object lookup table consumes 5.6 GB DRAM. For the same experiment, the other four systems recover faster than ArchTM. The overhead in recovery in ArchTM comes from scanning the PM data area because ArchTM needs to identify updates since the last checkpoint before the crash to rebuild the object lookup table. ArchTM trades a slightly longer recovery time for better runtime performance based on the assumption that crashes in the production environment are infrequent [10].

Transaction abort rate. Transaction aborts occur when a transaction tries to get the write lock of the writer of a persistent object but fails. We measure the abort rate. With 24 threads running highly write-intensive workloads with 80% update rate using the hash table and red-black tree, the abort rate is 1% and 2% on average, respectively. With 24 threads running the TPC-C and TATP, the abort rate is 2% and 2% on average, respectively. In general, the abort rate is very low.

7 Related Work

Undo-logging based PM transactions. Intel's PMDK [30] (libpmemobj) and NV-Heap [19] use undo-logging to log persistent objects on PM for crash recovery. Atlas [16] also uses undo-logging. It provides compiler and runtime supports to instrument writes to PM. JUSTDO logging [33] implements an Atlas-like log management system designed for machines with persistent caches. It stores the program counter and resumes the execution of critical sections from the same point where a crash happens. iDO [47] optimizes JUSTDO logging by avoiding logging each persistent store. Specifically, iDO divides the critical section into several idempotent code regions and only logs live program states at the beginning of each idempotent region within the critical section.

Redo-logging based PM transactions. NVthreads [29] supports redo-logging for multi-threaded C/C++ programs. It logs dirty pages tracked by the OS page protection between critical sections. DUDETM [46] uses shadow DRAM to decouple transaction updates and redo-logging. It leverages a background thread to copy and persist the modifications in redo logs to hide the logging overhead. Romulus [21] and Pisces [27] use variants of redo-logging. They both keep two copies of the data and replicate updates from one copy to the other to ensure crash consistency. Romulus uses a volatile log to record memory locations modified during a transaction to improve the performance of data copy. Pisces targets read-most workloads and explores snapshot isolation to ensure lock-free read operations.

CoW-based PM transactions. CDDS [68], BPFS [20], and multi-version concurrency control based transactions (e.g., TimeStone [40]) create a new copy and apply updates to the new copy to avoid writing log records.

The above logging-based and CoW-based works optimize PM transactions by reducing data replication or persistence overhead. In contrast, ArchTM introduces architecture-awareness to adapt the transaction system to leverage the micro-architecture (i.e., internal buffer and data size block) on the PM hardware. With the architecture-awareness, ArchTM improves the efficiency of PM writes by avoiding small writes and encouraging coalescable writes.

8 Conclusions

Enabling high-performance transactions is critical for leveraging persistent memory for data-intensive applications. We reveal performance problems in common transaction implementations on real PM hardware and highlight the importance of considering PM architecture characteristics for transaction performance. In this paper, we present ArchTM, an architecture-aware PM transaction system. On average, ArchTM outperforms the state-of-the-art PM transaction systems (PMDK, Romulus, DudeTM, and the Oracle system) by 58x, 5x, 3x, and 7x respectively.

Acknowledgment. This work was partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF1718194). This research was supported by the Exascale Computing Project (17-SC-20-SC). LLNL-CONF-808913. We thank our shepherd, Natacha Crook and anonymous reviewers for their constructive comments and suggestions.

References

[1] Key/value datastore for persistent memory. https://github.com/pmem/pmemkv.

[2] mongoDB. https://www.mongodb.com/.

[3] MySQL. https://www.mysql.com/.

[4] Oracle. www.oracle.com.

[5] PostgreSQL. https://www.postgresql.org/.

[6] Rdtscp — read time-stamp counter and processor id. www.felixcloutier.com/x86/rdtscp.

[7] Redis. https://redis.io/.

[8] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Technical report, USA, 1999.

[9] Aerospike. Building Real-Time Database at Petabyte Scale. https://www.aerospike.com/partners/intel-optane/.

[10] M. Alshboul, J. Tuck, and Y. Solihin. Lazy Persistency: A High-Performing and Write-Efficient Software Persistency Technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, June 2018.

[11] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endow.*, 10(4):337–348, November 2016.

[12] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, 1995.

[14] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, 2000.

[15] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677–694, October 2016.

[16] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014.

[17] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 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*, ASPLOS '20, 2020.

[18] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. Atmem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, 2020.

[19] 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 *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[20] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[21] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 271–282, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Christian Craft. Persistent Memory Primer. https://blogs.oracle.com/database/persistent-memory-primer.

[23] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. Technical report, 2006. http://jemalloc.net/.

[24] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[25] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory. *Proc. VLDB Endow.*, 13(10):1304–1318, April 2020.

[26] Wolfram Gloger. Wolfram Gloger's malloc. http://www.malloc.de/en/.

[27] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[28] Eric Hanson. How to Use MemSQL with Intel's Optane Persistent Memory. https://www.memsql.com/blog/how_to_use_memsql_with_intels_optane_persistent_memory.

[29] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482. ACM, 2017.

[30] Intel. Persistent Memory Development Kit. https://pmem.io/.

[31] Intel. ntel® Optane™ Persistent Memory 200 Series Delivers on Average 25% More Bandwidth with up to 4.5 TB Total Memory per Socket. https://newsroom.intel.com/wp-content/uploads/sites/11/2020/06/Optane-Mem-200-Series-Product-Brief.pdf.

[32] Intel. Revolutionizing Memory and Storage. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[33] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[35] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient Persist Barriers for Multicores. In *International Symposium on Microarchitecture*, 2015.

[36] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A scalable ordering primitive for multicore machines. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[37] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[38] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated Persist Ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[39] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[40] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[41] Redis Labs. Redis Enterprise on Intel® Optane™ DC Persistent Memory Offers Cost-Effective Scaling to Petabytes. https://redislabs.com/press/redis_enterprise_intel_optane_dc_persistent_memory_offers_cost_effective_scaling_petabytes_2.

[42] Scott T. Leutenegger and Daniel Dias. A Modeling Study of the TPC-C Benchmark. In *SIGMOD Record*, 1993.

[43] P. Li, D. R. Chakrabarti, C. Ding, and L. Yuan. Adaptive software caching for efficient nvram data persistence. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[44] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017.

[45] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. Si-tm: Reducing transactional memory abort rates through snapshot isolation. *SIGPLAN Not.*, 49(4):383–398, February 2014.

[46] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[47] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[48] S. Lu, A. Bernstein, and P. Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[49] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-Ordering Consistency for Persistent Memory. In *IEEE 32nd International Conference on Computer Design*, 2014.

[50] Virendra J. Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent memory transactions. *CoRR*, abs/1804.00701, 2018.

[51] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[52] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: Overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, 2019.

[53] Microsoft. Snapshot Isolation in SQL Server. https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server.

[54] Simo Neuvonen, Antoni Wolski, Markku manner, and Vilho Raatikka. Telecom Application Transaction Processing Benchmark . http://tatpbenchmark.sourceforge.net/.

[55] Lois Orosa and Rodolfo Azevedo. Logsi-htm: Log based snapshot isolation in hardware transactional memory. 07 2015.

[56] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[57] I. Peng, K. Wu, J. Ren, D. Li, and M. Gokhale. Demystifying the performance of hpc scientific applications on nvm-based memory systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 916–925, 2020.

[58] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*. ACM, 2019.

[59] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: Compacting memory management for c/c++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, 2019.

[60] L. Qiaoyu, L. Jianwei, and X. Yubin. Performance analysis of data organization of the real-time memory database based on red-black tree. In *2010 International Conference on Computing, Control and Industrial Engineering*, 2010.

[61] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, 2019.

[62] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[63] Torvald Riegel, Christof Fetzer, , and Pascal Felber. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, TRANSACT'06, 2006.

[64] Paul Menage Sanjay Ghemawat. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/.

[65] SAP. Realize the Promise of In-Memory Computing. https://discover.sap.com/sap-hana-dc-persistent-memorynew/en-us/index.html.

[66] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for nvram. In *ADMS@VLDB*, 2015.

[67] H. Shu, H. Chen, H. Liu, Y. Lu, Q. Hu, and J. Shu. Empirical Study of Transactional Management for Persistent Memory. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium*, 2018.

[68] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[69] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[70] H. Wan, Y. Lu, Y. Xu, and J. Shu. Empirical Study of Redo and Undo Logging in Persistent Memory. In *5th Non-Volatile Memory Systems and Applications Symposium*, 2016.

[71] Kai Wu, Ivy Peng, Jie Ren, and Dong Li. Ribbon: High performance cache line flushing for persistent memory. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, 2020.

[72] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 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, 2020.

[73] P. Zardoshti, T. Zhou, Y. Liu, and M. Spear. Optimizing persistent memory transactions. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[74] Pengfei Zuo, Yu Hua, and Jie Wu. Level hashing: A high-performance and flexible-resizing persistent hashing index structure. *ACM Trans. Storage*, 2019.