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

Kai Wu, Jie Ren
{kwu42,jren6}@ucmerced.edu
UC Merced

Ivy Peng
peng8@llnl.gov
LLNL

Dong Li
dli35@ucmerced.edu
UC Merced

1 INTRODUCTION

Failure-atomic transactions are a critical mechanism for accessing and manipulating data with crash consistency on persistent memory (PM). Extensive studies 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 or persistence overhead. 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 (i.e., Intel Optane DC PM) 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, Romulus [1], DUDETM [3], and an Oracle transaction system [2]), 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., when performing 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 may likely result in sequential 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 responds to allocation requests for specific data object sizes. Such size-class-based memory allocation is used to reduce memory fragmentation. However, it can allocate noncontiguous memory blocks to meet consecutive memory allocation requests because of the 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,

*The full paper of this abstract has been accepted into the 19th USENIX Conference on File and Storage Technologies (FAST'21), http://pasalabs.org/papers/2020/FAST21_ArchTM.pdf. LLNL-ABS-819593

ArchTM aggregates data objects in highly fragmented memory regions to create large and contiguous memory blocks.

In summary, 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 CoW-based 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.

2 MAJOR TECHNIQUES

1) Minimize metadata modifications on PM with guaranteed crash consistency. 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. 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 must modify the persistent object metadata on PM to update the pointer to the object to the new copy. With the metadata placed in DRAM, ArchTMreduces small PM writes and accelerates the lookup, but cannot ensure crash consistency.

ArchTM introduces an *annotation* mechanism to guarantee crash consistency. In particular, 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.

2) 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, the object lookup table in ArchTM resides on a contiguous DRAM space, which brings convenience for management (e.g., checkpointing) and migration. If 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.

3) 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 in ArchTM because sequential writes bring more benefits to small objects than large objects. ArchTM has two data paths for persistent object allocation and reclamation: (1) a regular data path used for large memory allocation and reclamation, similar to that in existing allocators like JEMalloc; and (2) a locality-aware data path for small allocations. The latter optimizes through a single free list and global recycling procedures.

A single free list is used in ArchTM for allocating objects of various sizes. Existing approaches 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. Using a single free list mitigates this problem.

Recycle and merge memory blocks globally. Current approaches 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.

4) Reduce Memory Fragmentation. Using a single free list for various allocation sizes could result in memory fragmentation. ArchTMuses 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 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. Also, the user-space solution is more flexible than offline static solutionsand can react to changes in the application during execution.

3 PERFORMANCE

We evaluate ArchTM against four state-of-the-art transaction systems (one in PMDK, Romulus [1], DudeTM [3], and one from Oracle [2]). ArchTM outperforms the competitor systems by 58x, 5x, 3x and 7x on average, using micro-benchmarks (i.e., hash table and red-black tree) and real-world workloads (i.e., TPC-C and TATP).

REFERENCES

[1] A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. SPAA '18, 2018.
[2] V. J. M. et al. Persistent memory transactions. *CoRR*, abs/1804.00701, 2018.
[3] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. Dudetm: Building durable transactions with decoupling for persistent memory. ASPLOS '17, 2017.