ARCHTM: ARCHITECTURE-AWARE, HIGH PERFORMANCE TRANSACTION FOR PERSISTENT MEMORY

Kai Wu, Jie Ren, Ivy Peng*, Dong Li

University of California, Merced
Lawrence Livermore National Laboratory*
Persistent Memory Architecture & Performance Characterization

One socket of the Optane PM-Architecture

Sequential and random write bandwidth at different write sizes on PM and DRAM
Write amplification happens if the write size does not match the Optane media transaction size.

- Small random writes on PM perform worse than sequential writes
  - When writing only 64 bytes, random write achieves merely 25% of the bandwidth of sequential write
- Random writes on PM have performance spikes at write sizes that are a multiple of 256 bytes, but random writes on DRAM exhibits no such pattern
- Write bandwidth of PM is significantly lower than that of DRAM

PM ≠ Slow DRAM
Insights of PM Performance Characterization

- PM microarchitecture (e.g., internal buffer and data block size) has significant impacts on performance
  - Avoid small random writes
  - Leverage the combining buffer to coalesce writes inside PM

- Reducing write traffic on PM is critical
Transactions on Persistent Memory

- Failure-atomic transaction is a critical mechanism for accessing and manipulating data on PM

- Existing PM transaction systems are implemented using two major paradigms – logging (undo & redo) and copy-on-write

😊 The implementation of both paradigms often does not consider the performance impact of PM architecture characteristics
Issues of Existing PM Transactions

**Undo-logging**
1. Copy data to logs
   - Data: A  B
   - Log: A

2. Data is updated in-place
   - Data: A  B
   - Log: A

3. Commit
   - Data: A  B

**Redo-logging**
1. Write updates to logs
   - Data: A  B
   - Log: A

2. Apply logs to the data
   - Data: A  B
   - Log: A

3. Commit
   - Data: A  B

- Data from last commit
- Newly written data
- The new copy of data
Issues of Existing PM Transactions

Undo-logging
1. Copy data to logs
   - Data: A, B
   - Log: A
2. Data is updated in-place
   - Data: A, B
   - Log: A
3. Commit
   - Data: A, B

Redo-logging
1. Write updates to logs
   - Data: A, B
   - Log: A
2. Apply logs to the data
   - Data: A, B
   - Log: A
3. Commit
   - Data: A, B

⚠️ Write data twice increases write traffic to PM
Issues of Existing PM Transactions

1. Allocate and initialize new copies
   - A
   - B
   - Ptr (A)
   -_PTR (B)

2. Write updates to new copies
   - A
   - B
   - Ptr (A)
   -_PTR (B)

3. Commit
   - A
   - B
   - Ptr (A)
   -_PTR (B)

Reset pointers and free old copy

Data from last commit
Newly written data
The new copy of data
Frequent metadata updates cause many small random writes
Issues of Memory Allocation for PM Transactions

- Existing memory allocation implementations use multiple free lists, each for a different allocation size
  - Consecutive memory allocation requests with different sizes can go to different free lists, leading to non-sequential memory accesses in the future

Noncontiguous memory blocks
Issues of Memory Allocation for PM Transactions

- Existing memory allocation implementations use multiple free lists, each for a different allocation size
  - Consecutive memory allocation requests with different sizes can go to different free lists, leading to non-sequential memory accesses in the future
  - Return freed memory blocks to thread-local free lists for reuse may harm the locality of the freed memory blocks and cause non-sequential memory access in the future

Reduce the opportunity to leverage the combining buffer to coalesce writes inside PM
Design Goals of ArchTM

- ArchTM: an architecture-aware PM transaction system
  - Reduce write traffic on PM
  - Avoid small writes on PM
  - Encourage coalescable writes on PM
  - Logless: Use copy-on-write
Avoid Small Writes on PM

- Minimize metadata modifications on PM with guaranteed crash consistency
Avoid Small Writes on PM

- Minimize metadata modifications on PM with guaranteed crash consistency
  - Buffer metadata on DRAM
    - Allocator metadata
    - Object mapping metadata
      - Object lookup table
Minimize metadata modifications on PM with guaranteed crash consistency

Buffer metadata on DRAM

Annotation

- Add transaction ID into the transaction state variable
- Add object metadata (e.g., Object ID, size, and transaction ID) into the object header

Buffering metadata on DRAM does not affect the failure recovery since the annotation mechanism sets up a connection between data objects and transactions, allowing the system to decide the consistency or validness of data objects.
Encourage Coalescable Writes on PM

- Consecutive allocation requests get contiguous memory blocks but minimize memory fragmentation
Encourage Coalescable Writes on PM

- Consecutive allocation requests get contiguous memory blocks but minimize memory fragmentation
  - Contiguous memory allocation
    - Use a regular data path for large objects
    - Use a locality-aware data path for small objects

Allocation or reclamation request → Small object? → No → Regular Data Path → Size class-based free lists → Thread-local recycling

Allocation or reclamation request → Small object? → Yes → Locality-aware Data Path → Single free list → Global recycling

Freed memory blocks will be sorted according to the memory addresses

Encourage sequential writes in transactions
Encourage Coalescable Writes on PM

- Consecutive allocation requests get contiguous memory blocks but minimize memory fragmentation
  - Contiguous memory allocation

- Online memory defragmentation
  - Examines memory usage by regions
  - Aggregate data in highly fragmented memory regions to create large and contiguous memory blocks

![Diagram of memory regions and underutilized region]

1. Allocate new memory
2. Migrate its data to the newly allocated memory

😊 Improve memory usage
Encourage Coalescable Writes on PM

- Locality-aware data path & online memory defragmentation

- Monitor the fragmentation ratio periodically

- Aggregate persistent objects in underutilized regions and migrates them to a newly allocated memory region
Step 1: detect uncommitted transactions

- Check the state of TX State variables
- Record the TXID of the uncommitted TXs

TX State Variables [TXID,State]

Recovery Manager

Uncommitted Tx ID Set

Object lookup table

DRAM

OBJ Header Data OBJ Header Data OBJ Header Data

PM
Step 2: Rebuild object lookup table

- Check the state of TX State variables
- Record the TXID of the uncommitted TXs
- Scan the persistent object pool to find persistent objects
- Insert the location information (i.e., pointers to the object on PM) into the object lookup table
- Discard the object copies involved in the uncommitted TXs
- Only keep the latest object copy
Other Optimization Techniques

- Fast object referencing through a scalable lookup table
- Maximize read concurrency by implementing lock-free reads
- Reduce recovery time by incorporating an incremental checkpoint

Please find more details in our paper!
Evaluation Setup

- Real PM platform (Intel Optane DC PMM)
  - 2nd Gen Intel Xeon Scabbled processor (24 cores on each socket)
  - 192 GB DRAM and 1.5 TB PM

- Run TPC-C and TATP against PMEMKV (from Intel)

- Comparison: PMDK [Intel], Romulus [SPAA’18], DUDETM [ASPLOS’17] and the Oracle system (copy-on-write-based, OCoW)
On average, ArchTM significantly outperforms DUDETM, Romulus, OCoW and PMDK by 3x, 7x, 8x and 75x, respectively.
Performance Contributions of Optimization Techniques

- Run TPC-C with 24 application threads
- Minimized metadata modification on PM contributes the most (66%) performance improvement

- Logless: apply CoW
- Minimized metadata modification (MMDPM): apply buffering metadata on DRAM and annotation
- Contiguous memory allocation (CMAlocation): apply locality-aware data path and online defragmentation
Conclusions

- Common logging- and CoW-based transaction implementations are not customized to real PM hardware.
- It is critical to consider PM architecture characteristics for high-performance transactions.
- **ArchTM**: an architecture-aware PM transaction system
  - Avoid small writes on PM
  - Encourage coalescable writes on PM
  - Outperform the four state-of-the-art PM transaction systems
Thank You

Q & A