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 (PM) Has Arrived

- Memory-like performance
  - ~100x faster than SSDs
  - Byte-addressability

- Storage-like characteristics
  - Non-volatility
  - High density
    - Each socket can have as much as 4.5 TB
PM Architecture & Performance Characterization

- Reducing write traffic on PM is critical
- PM microarchitecture (e.g., internal buffer and data block size) has a significant impact on the write performance of PM
  - Avoid small random writes
  - Leverage the combining buffer hardware to coalesce writes inside PM

Write bandwidth to DRAM reaches 60 GB/s but only 13 GB/s to Optane PM
Transactions on Persistent Memory

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

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

- Both paradigms do 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

#### Copy-on-Write
1. Allocate and initialize new copies
   - Data: A B
   - Log: A
2. Write updates to new copies
   - Data: A B
   - Log: A
3. Commit
   - Data: A B
   - Log: A
   - Reset pointers and free old copy

- X Data from last commit
- X Newly written data
- X 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

### Copy-on-Write
1. Allocate and initialize new copies
   - Data
     - A
     - B
   - Log
     - A
2. Write updates to new copies
   - Data
     - A
     - B
   - Log
     - A
3. Commit
   - Data
     - A
     - B

- Write data twice
- In-place update to the data could cause concurrent random writes
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

**Copy-on-Write**
1. Allocate and initialize new copies
   - Data: A  B
   - Log:  A
2. Write updates to new copies
   - Data: A  B
   - Log:  A
3. Commit
   - Data: A  B

- Write data twice
- In-place update to the data could cause concurrent random writes
- Frequent metadata updates causes many small random writes

*Reset pointers and free old copy*
Issues of Memory Allocation for PM Transactions

- Existing memory allocation implementations use multiple free lists, each for a different allocation size.

  1-2

  3

  4

- Multiple free lists could cause consecutive allocation requests of different sizes to go to different free lists.

- Return freed memory blocks to thread-local free lists for reuse.

Reduce the opportunity to leverage the combining buffer hardware 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
Avoid Small Writes on PM

- 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
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 allocations and reclamations
    - Use a locality-aware data path for small allocations and reclamations to encourage sequential writes in transactions
      - A single free list
      - Global recycling
  - Online memory defragmentation
    - Examines memory usage by regions and reduces fragmentation on PM during the runtime
Encourage Coalescable Writes on PM

- Locality-aware data path & online memory defragmentation

```
Global free list

Allocate
Private deallocation list
T1
Deallocate
Collect
GC manager

Allocate
Private deallocation list
T2
Deallocate
Collect
Defragmentation manager

Allocate
"Mock" write TX
Private deallocation list

Deallocate
Global recycle list

Monitor the fragmentation ratio periodically

Aggregate persistent objects in underutilized regions and migrates them to a newly allocated memory region
```

Refill

Collect
Recovery Management

- **Step 1: detect uncommitted transactions**
  - Check the state of each transaction state variable on PM

- **Step 2: rebuild object lookup table**
  - Scan persistent object pool on PM to find persistent objects
  - Insert the location information (i.e., pointers to the object on PM) into the lookup table
    - Discard the object copies in uncommitted transactions (collected from Step 1)
    - Only keep the latest object copy by comparing the transaction ID annotated in the object copies
Other Optimization Techniques

- Scalable object referencing
- Non-blocking read
- 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 Scabble 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 DUDET, Romulus, OCoW and PMDK by 3x, 7x, 8x and 75x, respectively.

Please find more evaluation in our paper!
Conclusion

- Pinpoint performance problems in common transaction implementations on real PM hardware

- Highlight the importance of considering PM architecture characteristics for transaction performance

- 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