Loose-Ordering Consistency for Persistent Memory

Youyou Lu[1], Jiwu Shu[1], Long Sun[1], Onur Mutlu[2]

[1]Tsinghua University
[2]Carnegie Mellon University

Summary

- Problem: Strict write ordering required for storage consistency dramatically degrades performance in persistent memory
- Our Goal: To keep the performance overhead low while maintaining the storage consistency
- Key Idea: To Loosen the persistence ordering with hardware support
  - Eager commit: A commit protocol that eliminates the use of commit record, by reorganizing the memory log structure
  - Speculative persistence: Allows out-of-order persistence to persistent memory, but ensures in-order commit in programs, leveraging the tracking of transaction dependencies and the support of multi-versioning in the CPU cache
- Results: Reduces average performance overhead of persistence ordering from 67% to 35%

Outline

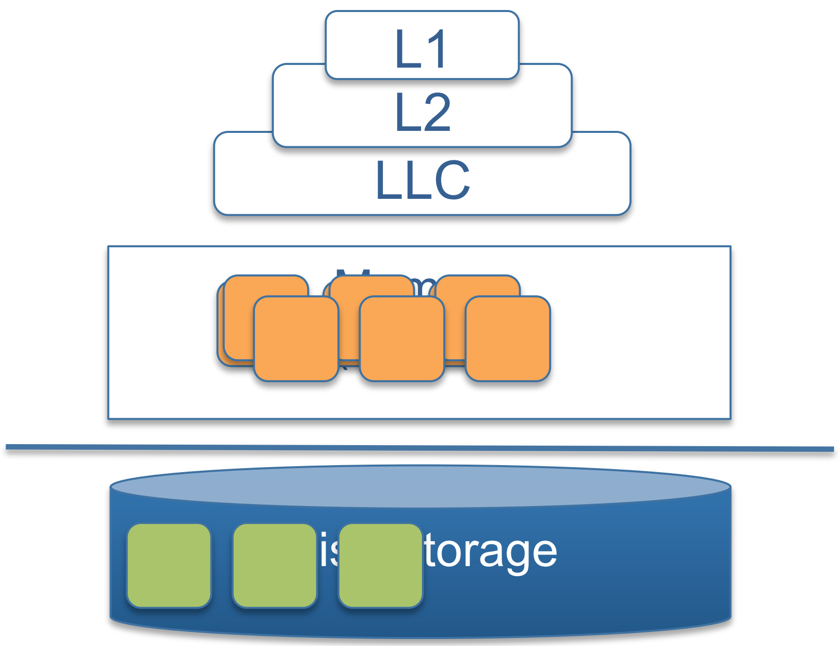
- Introduction and Background
- Existing Approaches
- Our Approach: Loose-Ordering Consistency
  - Eager Commit
  - Speculative Persistence
- Evaluation
- Conclusion

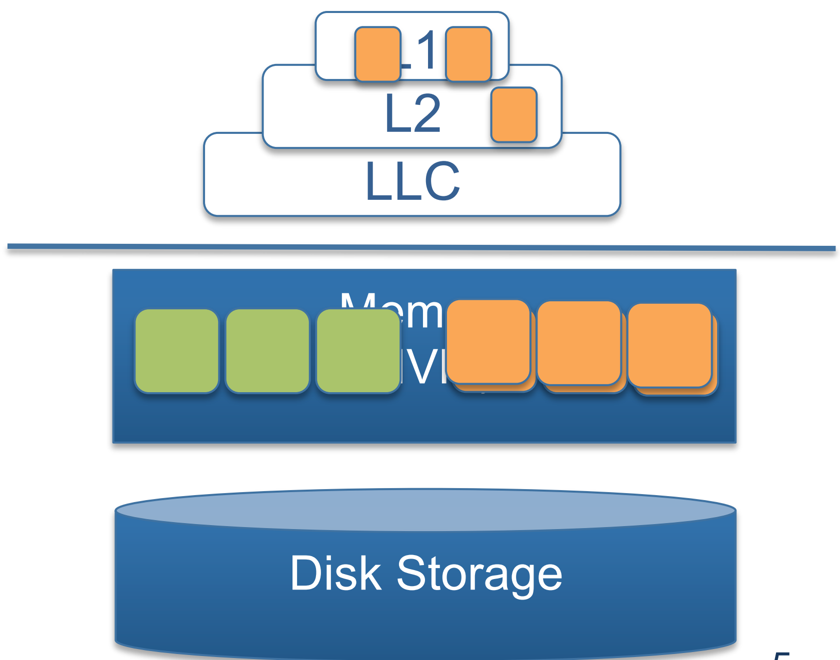
Outline

- Introduction and Background
- Existing Approaches
- Our Approach: Loose-Ordering Consistency
  - Eager Commit
  - Speculative Persistence
- Evaluation
- Conclusion

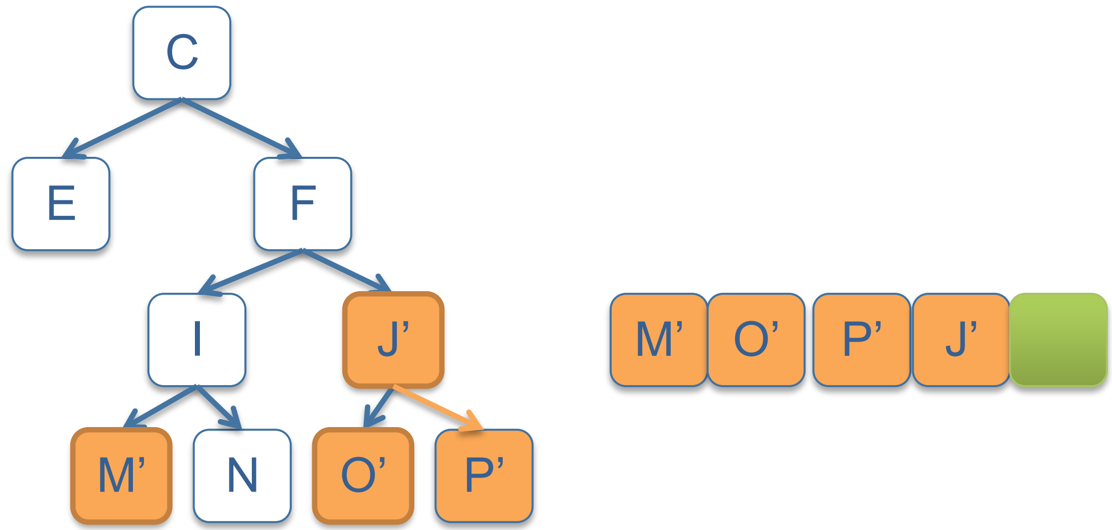
Persistent Memory

- Persistent Memory
  - Memory-level storage: Use non-volatile memory in main memory level to provide data persistence
- Storage Consistency
  - Atomicity and Durability: Recoverable from unexpected failures
  - Boundary of volatility and persistence moved from Storage/ Memory to Memory/Cache





Storage Consistency – Write-Ahead Logging(WAL)



- Step 1. Log Write
- Step 2. Commit Record Write
- Step 3. In-place Write
- Step 4. Log Truncation

Intra-tx Ordering

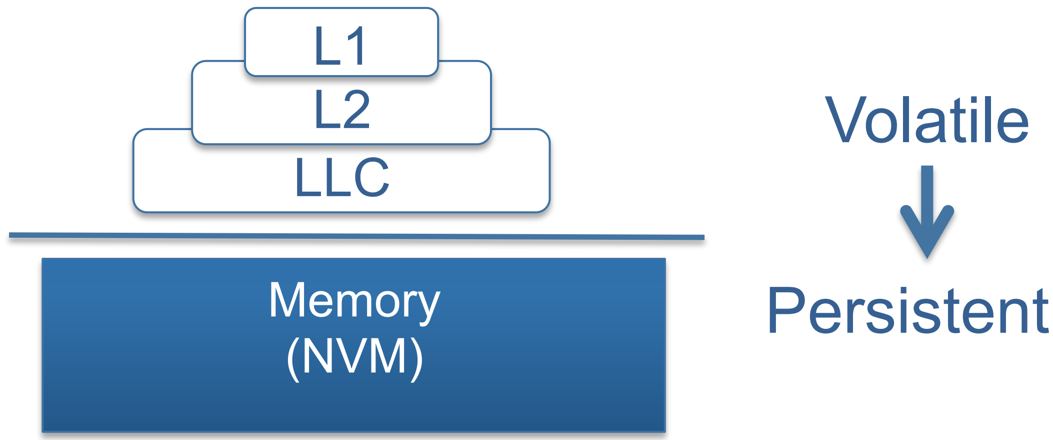
Program Ack

Inter-tx Ordering

Ordering is required for storage consistency.

High Overhead for Ordering in PM

- Persistence ordering
  – Force writes from volatile CPU cache to Persistent Memory



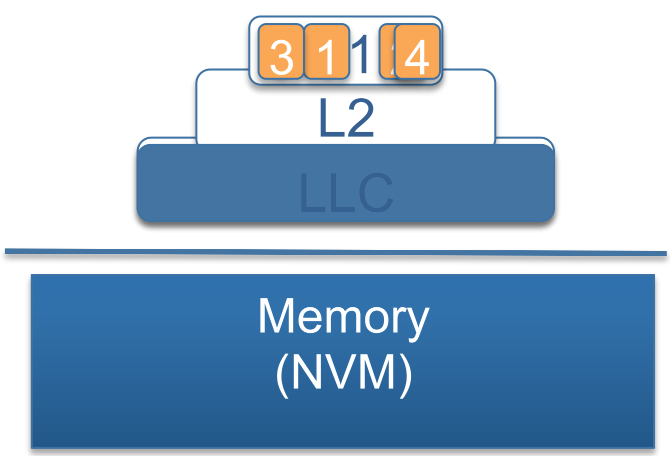
- High overhead for persistence ordering
  – The boundary between volatility and persistence lies between the H/W controlled cache and the persistent memory
    - Costly software flushes (*clflush*) and waits (*fence*)
  – Existing systems reorder writes at multiple levels, especially in the CPU and cache hierarchy

Outline

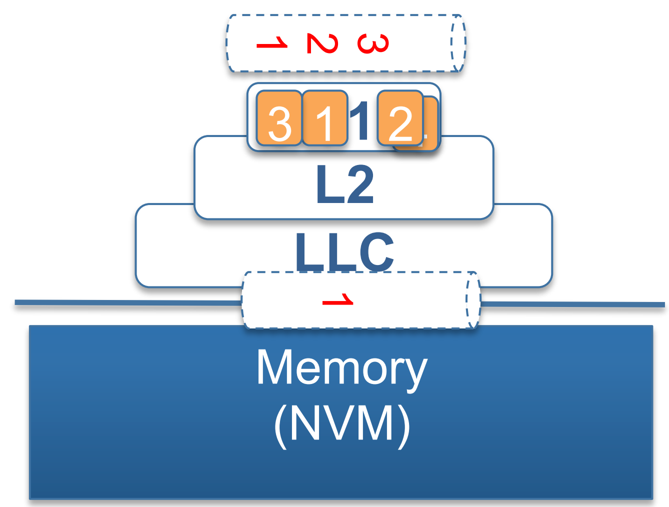
- Introduction and Background
- Existing Approaches
- Our Approach: Loose-Ordering Consistency
  - Eager Commit
  - Speculative Persistence
- Evaluation
- Conclusion

Existing Approaches

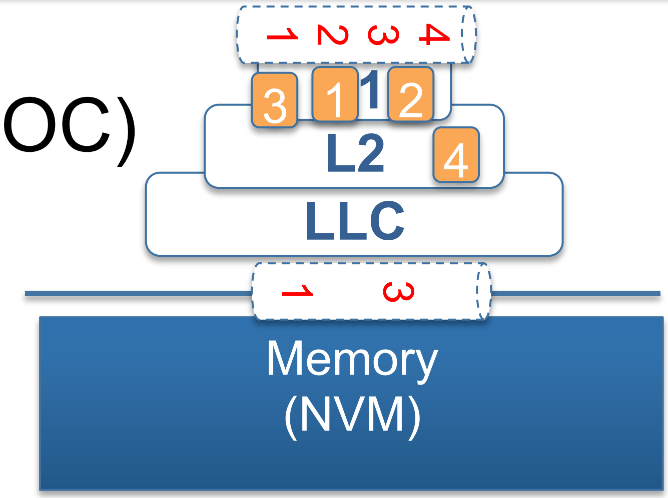
- Making the CPU cache non-volatile
  - Reduce the time gap between volatility and persistence by employing a non-volatile cache
  - Is complementary to our LOC approach
- Allowing asynchronous commit of transactions
  - Allow the execution of a later transaction without waiting for the persistence of previous transactions
  - Allow execution reordering, but no persistence reordering



T1: A, B, C, D
T2: A, F
T3: B, C, E
T4: D, E, F, G



Our Solution: Key Ideas



- Loose-Ordering Consistency (LOC)
  - Allow persistence reordering
- Eager Commit
  - Remove the intra-tx ordering
    - Delay the completeness check till recovery phase
  - Reorganize the memory log structure
- Speculative Persistence
  - Relax the inter-tx ordering
    - Speculatively persist transactions but make the commit order visible to programs in the program order
  - Use cache versioning and Tx dependency tracking

Outline

- Introduction and Background
- Existing Approaches
- Our Approach: Loose-Ordering Consistency
  - Eager Commit
  - Speculative Persistence
- Evaluation
- Conclusion

LOC Key Idea 1 – Eager Commit

- Step 1. Log Write
- ~~Step 2. Commit Record Write~~
- Step 3. In-place Write
- Step 4. Log Truncation

~~Intra-tx Ordering~~

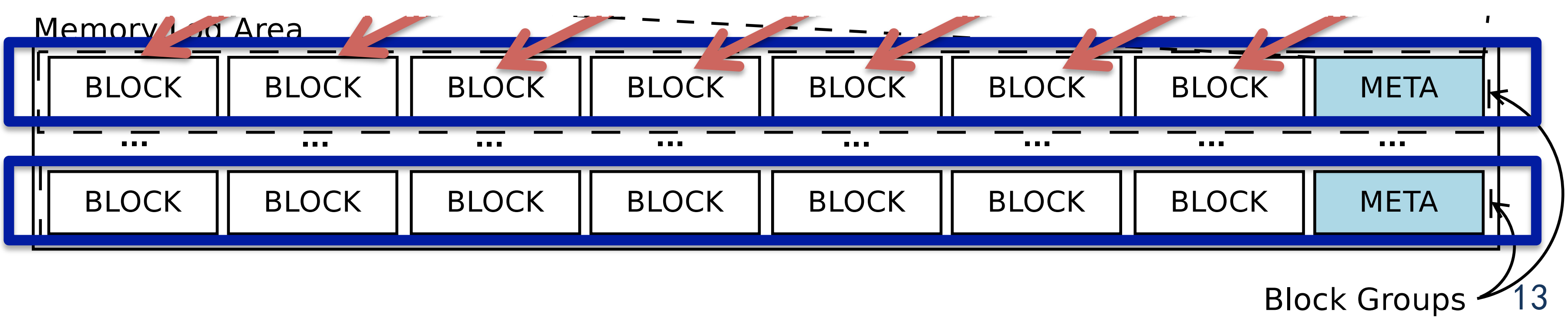
Program Ack

Inter-tx Ordering

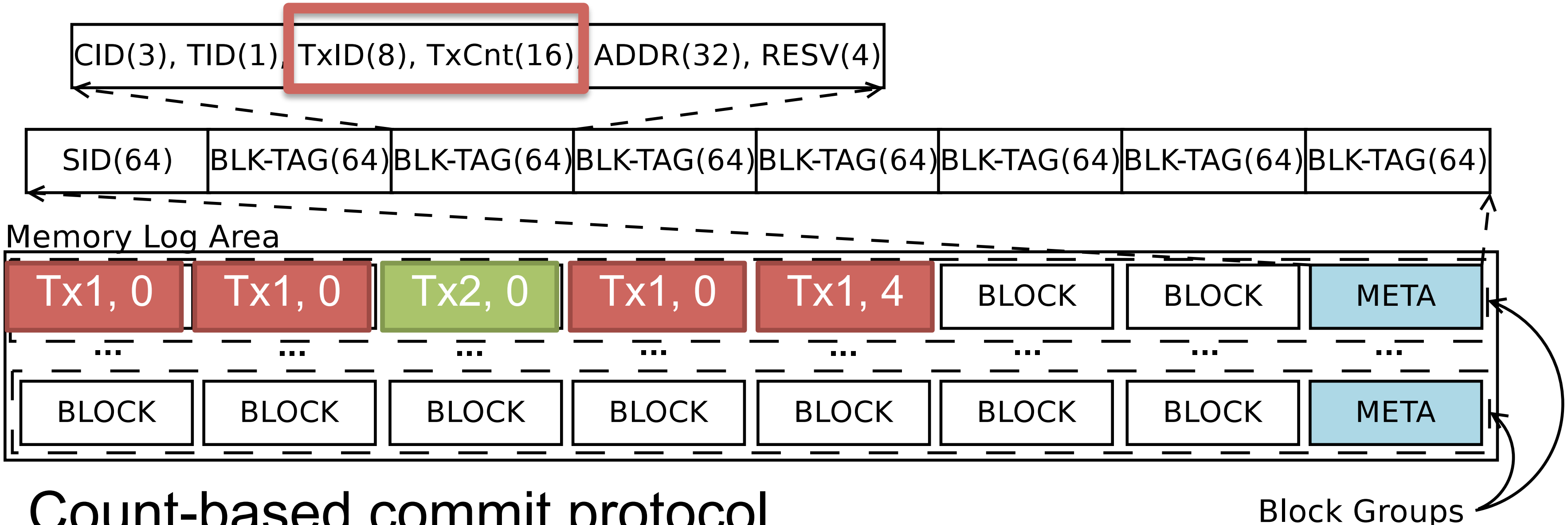
- Goal: Remove the intra-tx ordering
- Eager Commit: A new commit protocol without commit records

Eager Commit

- Commit Protocol
  - Commit record: Check the completeness of log writes
- Eager Commit
  - Reorganize the memory log structure for delayed check
    - Remove the commit record and the intra-tx ordering
  - Use count-based commit protocol: <TxID, TxCnt>



Eager Commit



- Count-based commit protocol
  - During normal run,
    - Tag each block with TxID
    - Set only one TxCnt to the total # of blocks in the tx, and others to '0'
  - During recovery,
    - Recorded TxCnt: Find the non-zero TxCnt for each tx TxID
    - Counted TxCnt: Count the tot. # of blocks in the tx
    - If the two TxCnts match (Recorded = Counted), committed; otherwise, not-committed

No commit record. Intra-tx ordering eliminated.

LOC Key Idea 2 – Speculative Persistence

- Step 1. Log Write
- Step 2. Commit Record Write
- Step 3. In-place Write
- Step 4. Log Truncation

Intra-tx Ordering

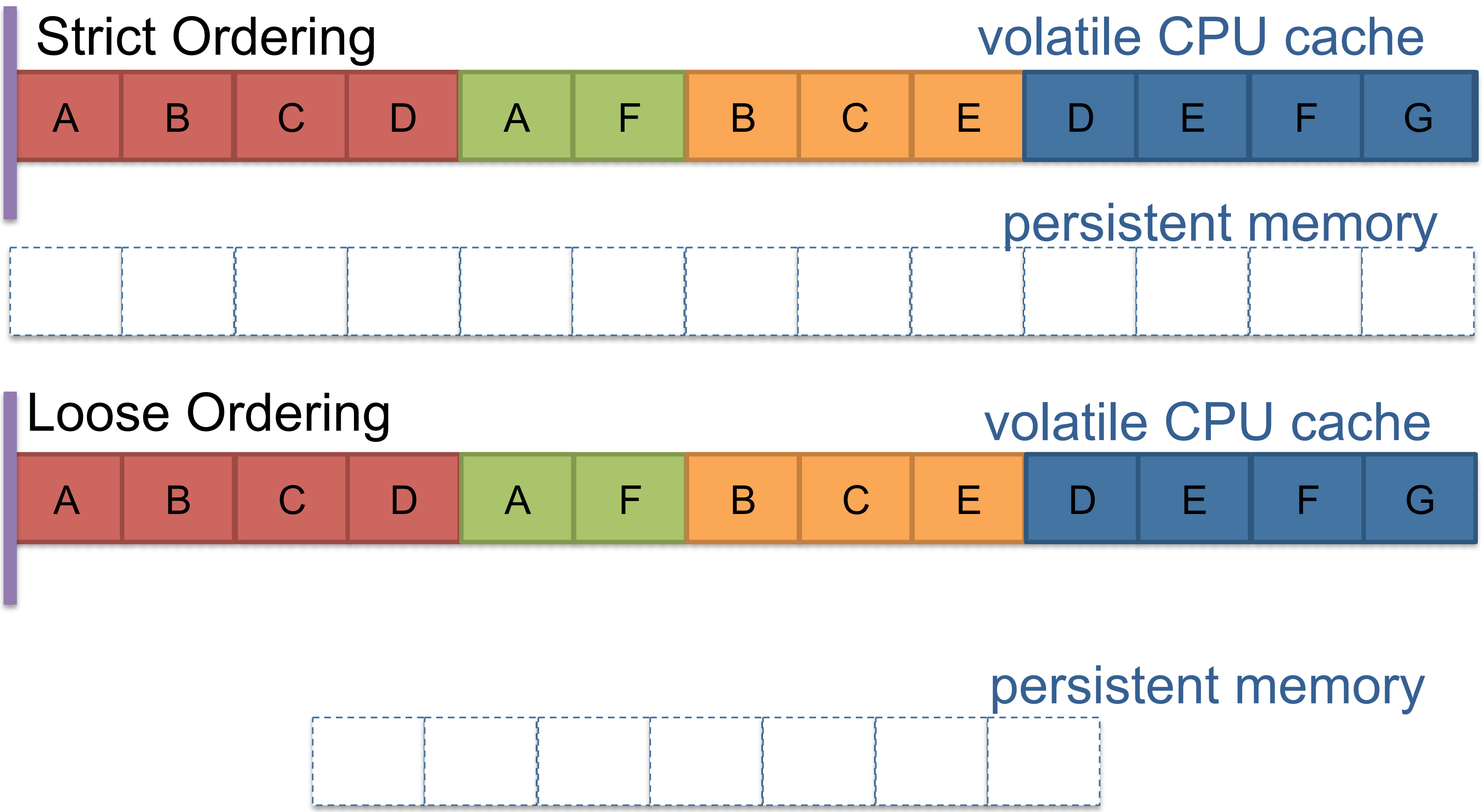
Program Ack

Inter-tx Ordering

- Goal: relax the inter-tx ordering
- Speculative Persistence
  - Out-of-order persistence: To relax the inter-tx ordering to allow persistence reordering
  - In-order commit: To make the tx commits visible to programs (program ack) in the program order

Speculative Persistence

T1: (A, B, C, D) -> T2: (A, F) -> T3: (B, C, E) -> T4: (D, E, F, G)



Inter-tx ordering relaxed. Write coalescing enabled.

Speculative Persistence

- Speculative Persistence enables write coalescing for overlapping writes between transactions.
- But there are two problems raised by write coalescing of overlapping writes:
  - How to recover a committed Tx which has overlapping writes with a succeeding aborted Tx?
    - Overlapping data blocks have been overwritten
  - Multiple Versions in the CPU Cache
  - How to determine the commit status using the count-based commit protocol of a Tx that has overlapping writes with succeeding Txs?
    - Recorded TxCnt != Counted TxCnt
  - Commit Dependencies between Transactions
    - Tx Dependency Pair: <Tp, Tq, n>

See the paper for more details.

Recovery

- Recovery is made by scanning the memory log.
- More details in the paper.

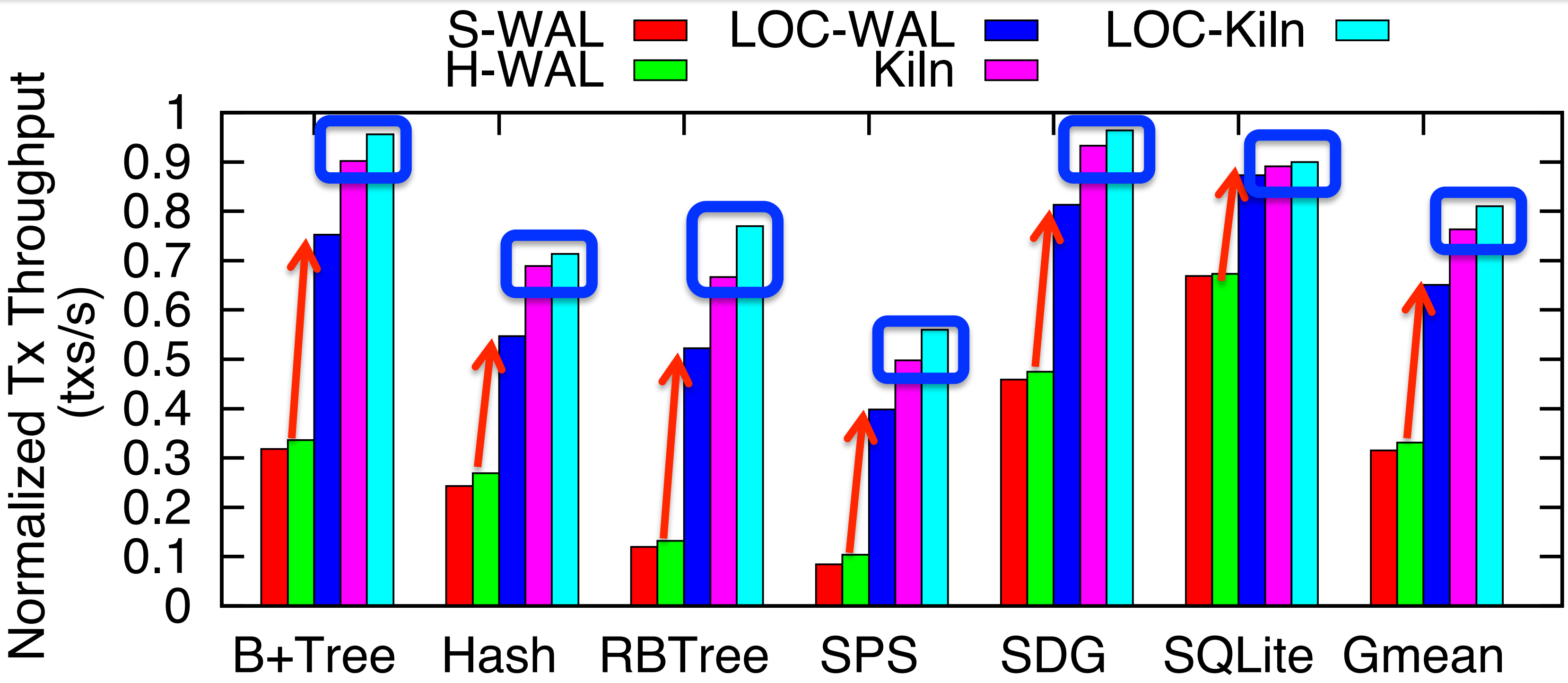
Outline

- Introduction and Background
- Existing Approaches
- Our Approach: Loose-Ordering Consistency
  - Eager Commit
  - Speculative Persistence
- Evaluation
- Conclusion

Experimental Setup

- GEM5 simulator
  - Timing Simple CPU: 1GHz
  - Ruby memory system
- Simulator configuration
  - L1: 32KB, 2-way, 64B block size, latency=1cycle
  - L2: 256KB, 8-way, 64B block size, latency=8cycles
  - LLC: 1MB, 16-way, 64B block size, latency=21cycles
  - Memory: 8 banks, latency=168cycles
- Workloads
  - B+ Tree, Hash, RBTree, SPS, SDG, SQLite

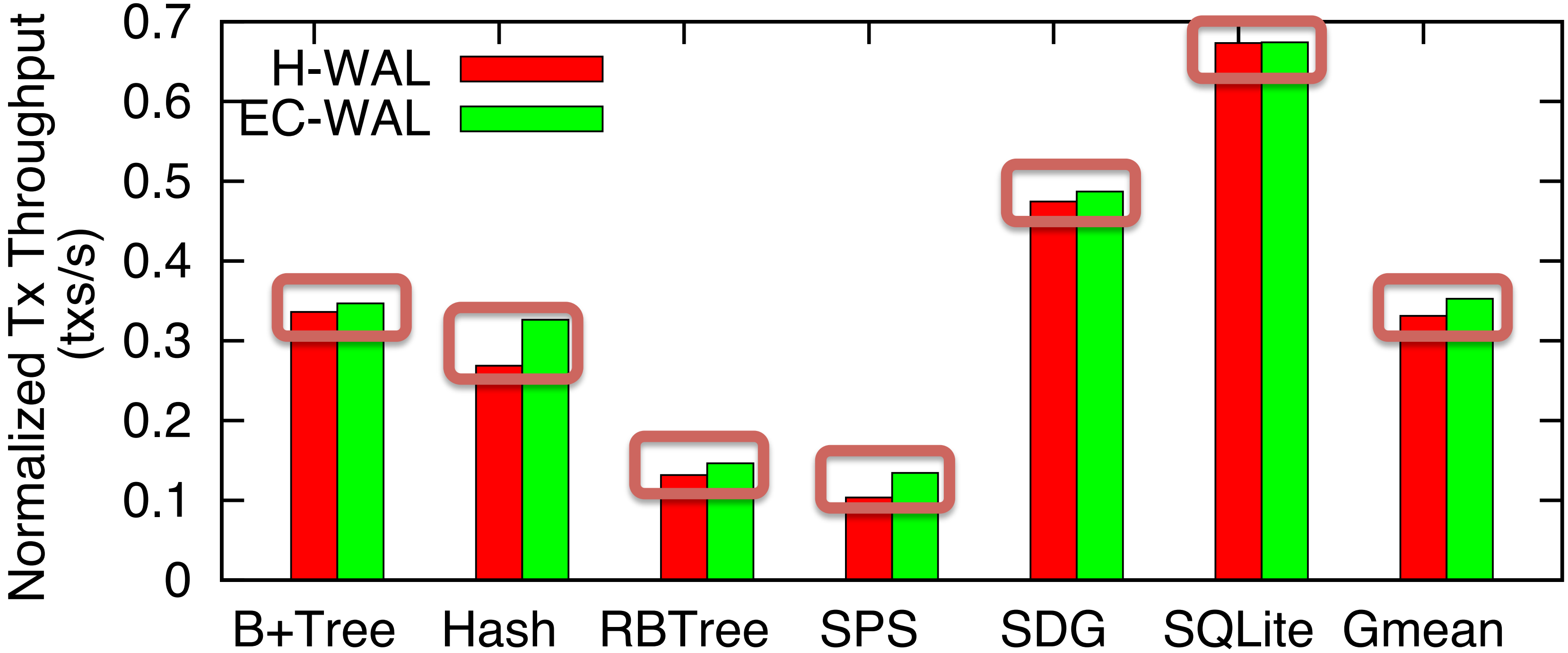
Overall Performance



- LOC significantly improves performance of WAL: Reduces average performance overhead of persistence ordering from 67% to 35%.
- LOC and Kiln can be combined favorably.

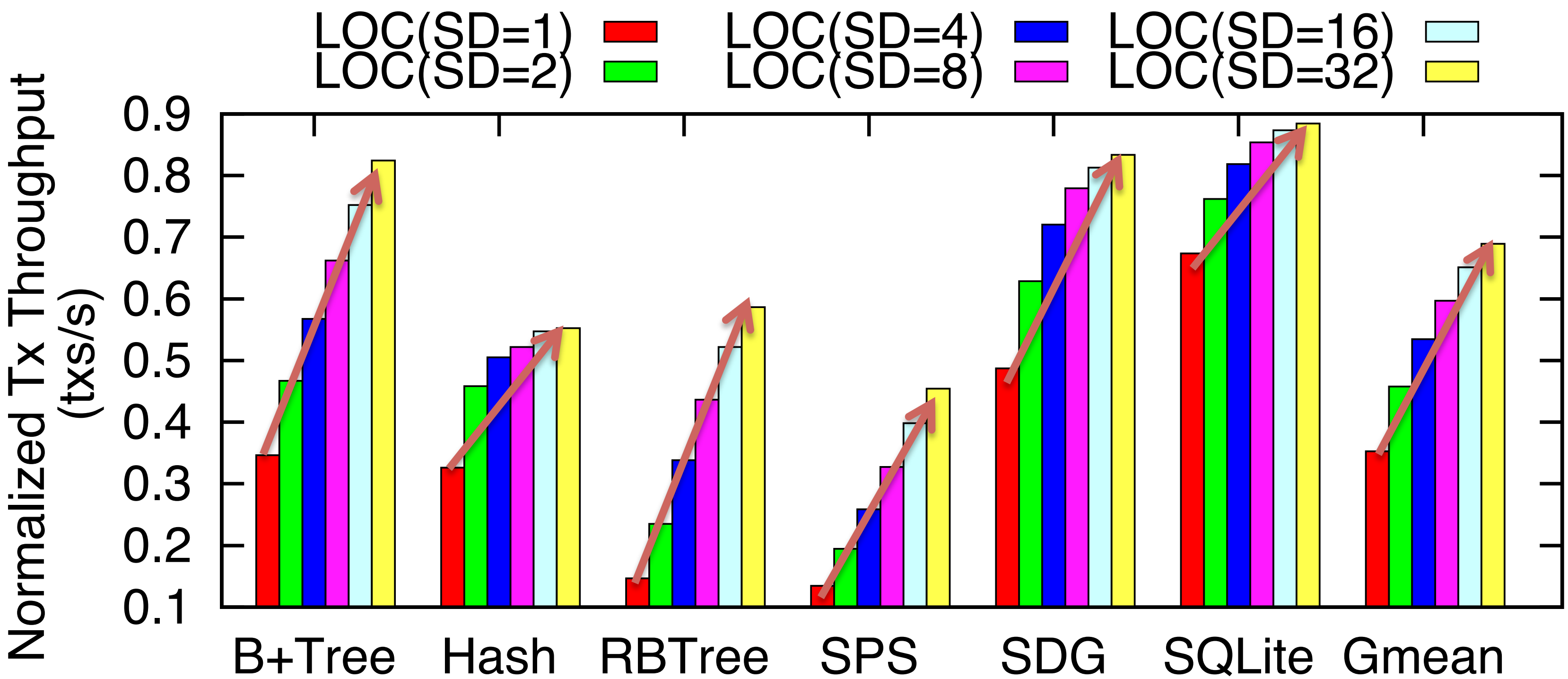
LOC effectively mitigates performance degradation from persistence ordering.

Effect of Eager Commit



Eager Commit outperforms H-WAL by 6.4% on average due to the elimination of intra-tx ordering.

Effect of Speculative Persistence



The larger the speculation degrees, the larger the performance benefits.

Speculative Persistence improves the normalized transaction throughput from 0.353 (SD=1) to 0.689 (SD=32) with a 95.5% improvement.

Outline

- Introduction and Background
- Existing Approaches
- Our Approach: Loose-Ordering Consistency
  - Eager Commit
  - Speculative Persistence
- Evaluation
- Conclusion

Conclusion

- Problem: Strict write ordering required for storage consistency dramatically degrades performance in persistent memory
- Our Goal: To keep the performance overhead low while maintaining the storage consistency
- Key Idea: To Loosen the persistence ordering with hardware support
  - Eager commit: A commit protocol that eliminates the use of commit record, by reorganizing the memory log structure
  - Speculative persistence: Allows out-of-order persistence to persistent memory, but ensures in-order commit in programs, leveraging the tracking of transaction dependencies and the support of multi-versioning in the CPU cache
- Results: Reduces average performance overhead of persistence ordering from 67% to 35%

Loose-Ordering Consistency for Persistent Memory

Youyou Lu[1], Jiwu Shu[1], Long Sun[1], Onur Mutlu[2]

[1]Tsinghua University
[2]Carnegie Mellon University