A Fast, General System for Buffered Persistent Data Structures

Haosen Wen*, Wentao Cai*, Mingzhe Du, Louis Jenkins, Benjamin Valpey and Michael L. Scott

University of Rochester

ICPP '21

*Equal contributions
Background

• Non-volatile memory (NVM) offers the possibility of keeping pointer-rich data structures across program runs and even crashes:
  • Correct persistence order is needed for crash consistency
  • Volatile caches mean that stores may reach memory out of program order; explicit write-back and fence instructions are necessary
  • Durable linearizability [Izraelevitz et al., DISC’16] necessitates high latency in every operation—ops must persist before returning
  • Buffered durable linearizability might reduce this latency, but all known implementations are ad-hoc
Montage

• First general-purpose system for buffered durably linearizable data structures

• Excellent performance, makes good use of NVM by:
  • Persisting periodically (every 1 – 10ms, or whenever sync() is called) rather than per-operation
  • Persisting only abstract data
Persistence Order: Durable Linearizability

• **Durable Linearizability** [Izraelevitz et al., DISC'16] :
  • Intuitive correctness criterion: operations persist before return
  • Enforced by writes-back (for persistence) and fences (for ordering) on *every* happens-before relationship on persistent data
  • Significant overhead
**Buffered Durable Linearizability**

- **Buffered Durable Linearizability** [Izraelevitz et al., DISC'16]:
  - After a crash, drop not-fully-persisted suffix of the history
  - Just make sure if $O_1$ happens before $O_2$ and $O_2$ is persisted, $O_1$ must be persisted
  - Agrees with persistency models of databases and file systems

- Reduces the overhead of persistence ordering
  - Avoid the need to write back and fence each op before returning & on each happens-before relationship
Montage: Periodic Persistence

• Inspired by Dalí [Nawab et al., DISC’17], Montage implements **buffered** durable linerizability by dividing time into *epochs*, and

\[
\text{epoch}(O_1) < \text{epoch}(O_2) \Rightarrow \neg (O_2 <_{hb} O_1)
\]

• Each operation is marked with *one* epoch

• Operations in the same epoch persist together, atomically
Montage: Periodic Persistence

• Design:
  • Write operations are assigned epoch numbers
    • All writes of an operation are marked with the same epoch
Montage: Periodic Persistence

• Design:
  • Write operations are assigned epoch numbers
    • All writes of an operation are marked with the same epoch
Montage: Periodic Persistence

• Design:
  • Write operations are assigned epoch numbers
    • All writes of an operation are marked with the same epoch
Montage: Periodic Persistence

• Design:
  • Before $e \rightarrow e + 1$, operations in $e - 1$ are finished and persisted
Montage: Periodic Persistence

• Design:
  • Before $e \rightarrow e + 1$, operations in $e - 1$ are finished and persisted
Montage: Periodic Persistence

• Design:
  • If we crash in $e$, all operations in $e - 1$ and $e$ are discarded
  • The boundary between $e - 2$ and $e - 1$ is chosen as the consistent cut
  • No in-place updates of blocks from old epochs – copy to preserve history
Montage: Periodic Persistence

• Design:
  • If we crash in $e$, all operations in $e - 1$ and $e$ are discarded
  • The boundary between $e - 2$ and $e - 1$ is chosen as the consistent cut
  • No in-place updates of blocks from old epochs – copy to preserve history
Montage: Periodic Persistence

• Design:
  • If we crash in $e$, all operations in $e - 1$ and $e$ are discarded
    • The boundary between $e - 2$ and $e - 1$ is chosen as the consistent cut
    • No in-place updates of blocks from old epochs – copy to preserve history
Montage: Periodic Persistence

• Design:
  • If we crash in \( e \), all operations in \( e - 1 \) and \( e \) are discarded
    • The boundary between \( e - 2 \) and \( e - 1 \) is chosen as the consistent cut
    • No in-place updates of blocks from old epochs – copy to preserve history
Montage: Periodic Persistence

• Design:
  • If we crash in $e$, all operations in $e - 1$ and $e$ are discarded
  • The boundary between $e - 2$ and $e - 1$ is chosen as the consistent cut
  • No in-place updates of blocks from old epochs – copy to preserve history
Montage: Periodic Persistence

• Design:
  • If we crash in $e$, all operations in $e - 1$ and $e$ are discarded
    • The boundary between $e - 2$ and $e - 1$ is chosen as the consistent cut
    • No in-place updates of blocks from old epochs – copy to preserve history
Montage: Periodic Persistence

• Design:
  • If we crash in $e$, all operations in $e - 1$ and $e$ are discarded
    • The boundary between $e - 2$ and $e - 1$ is chosen as the consistent cut
    • No in-place updates of blocks from old epochs – copy to preserve history

\[ O_1 = W_1 \quad O_1' = W_6 \]
Montage: Periodic Persistence

• Design:
  • Data structure must ensure each operation linearizes in the epoch of its writes
    • Operation in $E_1$ seeing blocks from $E_2 > E_1$ suggests there might be a problem. Montage (optionally) raises an exception to help
Montage: Periodic Persistence

• Design:
  • Data structure must ensure each operation linearizes in the epoch of its writes
    • Operation in $E_1$ seeing blocks from $E_2 > E_1$ suggests there might be a problem. Montage (optionally) raises an exception to help

$$O_1 = W_6$$

$W_6$ is from $E_2$!
Are you sure?
Montage: Persisting Abstract Data Only

• Inspired by NV-Tree [Yang et al., FAST'15], FPTree [Oukid et al., SIGMOD'16], Ralloc [Cai et al., ISMM'20], and Pronto [Memaripour et al., ASPLOS'20], among others, data structures can be rebuilt from abstract data after a crash
  • Sets/maps: keys (and values)
  • Queues: values and order
  • Graphs: vertices and edges

• Abstract data may comprise the majority of data structure's memory
• Can always persist more than abstract data for faster recovery
Montage Persistent Mapping

Only persist abstract data

- Root
- (Node)
  - (Node)
    - (Node)
      - "A", 1
      - "B", 2
    - (Node)
      - "C", 3
    - (Node)
      - "D", 4
Montage: Implementation

- Use Ralloc [Cai et al., ISMM'20] as NVM allocator
- Montage provides (C++) API to:
  - track reads and writes ((de-)allocations, updates) from/to persistent payloads.
  - identify the boundaries of each operation to ensure writes are marked with the same epoch for an operation
Montage: Implementation

• Persisting writes, buffering reclamations:
  • clwb right after each write messes up cache locality on current machines, while buffering unbounded writes brings overhead and stretches epochs
  • Bounded buffers for to-be-persisted writes
  • Reclamations must be buffered for 2 epochs – cannot be undone after crash
  • Only need those containers for 4 epochs: reuse containers from 3 epochs ago
Montage: Implementation

- Epoch advances and sync()
  - Epoch advances every 1 – 10 ms, automatically
  - Epoch $e$ gets persisted in $e + 2$, so sync() asks epoch to advance twice immediately
  - sync() blocks until all returned operations persists, for safe external communication
    - All sync() participants help write-back, coordinated under tree-structured mechanism
  - A background epoch advancer thread, before advancing to $e + 1$:
    - Reclaim payloads from $e - 2$
    - Complete writes-back for $e - 1$
    - sfence
    - Advance epoch
    - If sync() ongoing, repeat until all sync() goals are met
  - Superior performance even with sync() after every operation
Hash map performance (y log scale)

Hash Maps: 90% lookups, 10% updates
Conclusion

• Montage reduces the persistence overhead of recoverable data structures by:
  • Reducing cost of persist ordering
  • Reducing the amount of persistent data
• Suitable for both lock-based and nonblocking data structures
• Unprecedented performance
• Successor: Fast Nonblocking Persistence for Concurrent Data Structures, DISC’21
• Future work: Atomic composition of operations on multiple data structures
• Artifact: https://github.com/urcs-sync/Montage