Brief Announcement: Building Fast Recoverable Persistent Data Structures with Montage

Haosen Wen[1]
University of Rochester, NY, USA
hwen5@cs.rochester.edu

Wentao Cai[1]
University of Rochester, NY, USA
wcai5@cs.rochester.edu

Mingzhe Du
University of Rochester, NY, USA
mdu5@cs.rochester.edu

Benjamin Valpey
University of Rochester, NY, USA
bvalpey@cs.rochester.edu

Michael L. Scott
University of Rochester, NY, USA
scott@cs.rochester.edu

Abstract

The recent emergence of fast, dense, nonvolatile main memory suggests that certain long-lived data structures might remain in their natural, pointer-rich format across program runs and hardware reboots. Operations on such structures must be instrumented with explicit write-back and fence instructions to ensure consistency in the wake of a crash. Techniques to minimize the cost of this instrumentation are an active topic of current research.

We present what we believe to be the first general-purpose approach to building *buffered durably linearizable* persistent data structures, and a system, Montage, to support that approach. Montage is built on top of the Ralloc nonblocking persistent allocator. It employs a slow-ticking *epoch clock*, and ensures that no operation appears to span an epoch boundary. If a crash occurs in epoch e, all work performed in epochs e and $e-1$ is lost, but all work from prior epochs is preserved.

We describe the implementation of Montage, argue its correctness, and report on experiments confirming excellent performance for operations on queues, sets/mappings, and general graphs.

2012 ACM Subject Classification Theory of computation → Parallel computing models; Computing methodologies → Concurrent algorithms; Computer systems organization → Reliability

Keywords and phrases Durable linearizability, consistency, persistence, fault tolerance

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.52

Related Version `https://arxiv.org/abs/2009.13701`

Supplementary Material `https://github.com/urcs-sync/Montage`

Funding This work was supported in part by NSF grants CCF-1717712 and CNS-1900803, and by a Google Faculty Research award.

1 Background and Overview

A data structure that is intended to survive system crashes is generally expected to be *durably linearizable* [3], meaning (1) it is linearizable during crash-free operation, (2) each operation persists (reaches a state that will survive a crash) between its invocation and response, and (3) the order of persists matches the linearization order. At the cost of locality (trivial composability), a *buffered* durably linearizable data structure may preserve only some consistent prefix of its happens-before history on a crash.

Publications over the past few years have described some two dozen general-purpose systems to provide failure atomicity for outermost critical sections or speculative transactions on a concurrent data structure. The need for operations to persist before returning is a significant source of overhead in these systems. As an example of how one might reduce this overhead, Nawab et al. presented a buffered durably linearizable hash table (Dalí [5])

[1] The first two authors (Haosen Wen and Wentao Cai) contributed equally to this paper.

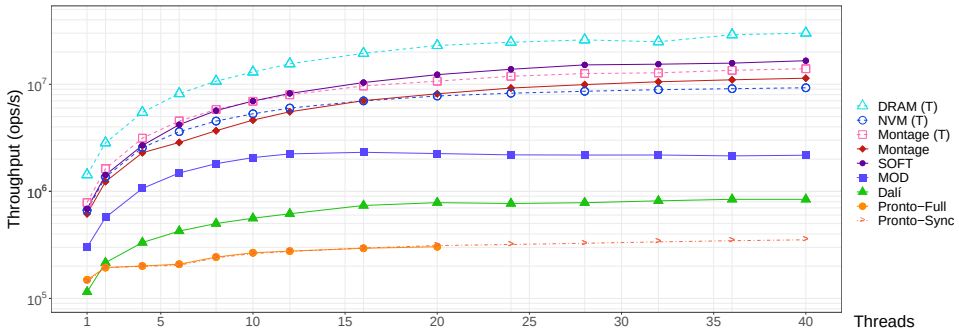

Figure 1 Persistent mapping throughput – 2:1:1 get:insert:remove; T = transient only.

that persists on a *periodic* (as opposed to incremental) basis. More recently, Haria et al.'s MOD project [2] proposed that in history-preserving ("functional") tree structures, each update can be persisted by updating a single root pointer, eliminating the need for logging. Concurrently, Memaripour et al.'s Pronto project [4] proposed that concurrent objects log their *high level* (abstract) operations (rather than low-level updates), together with occasional checkpoints; on a crash, they can then replay the suffix of the log past the most recent checkpoint. Similarly, Zuriel et al.'s SOFT [6] keeps the abstract state of a set – its keys and values – in both DRAM and NVM; the NVM copy is used only for recovery.

Inspired in part by these previous projects, we present what we believe to be the first general-purpose approach to *buffered* durably linearizable structures. Our system, Montage, preserves the *abstract* state of the concurrent object without necessarily persisting its *concrete* state. In a mapping, for example, it persists only a bag of key-value pairs; the look-up structure (hash table, tree, skip list) lives entirely in transient DRAM. During normal (crash-free) execution, Montage employs a slow-running *epoch clock*, and ensures that no operation appears to span an epoch boundary. If a crash occurs in epoch e, Montage recovers the state of the abstraction from the end of epoch $e-2$ and rebuilds the concrete structure.

Our implementation of Montage is built on top of Ralloc [1], a lock-free allocator for persistent memory. Montage itself is also lock-free during normal operation, except for epoch advances: if the data structure is itself nonblocking, a stalled thread will not impede operations of its peers; it can, however, indefinitely prevent those operations from persisting.

2 Implementation and Correctness

A typical concurrent object in Montage consists of a collection of *payload* blocks in nonvolatile memory, together with an index or other supporting structure in transient memory (DRAM). A global *epoch clock* is also kept in persistent memory. Each payload block indicates the epoch in which it was created. During recovery from a crash in epoch e, all blocks created in epochs e and $e-1$ are discarded in Ralloc. A block that was created in epoch e can be modified *in place* in epoch e (under protection of whatever synchronization would normally be used for the concurrent object). A block created in epoch $d < e$ is not modified in place; rather, a new block is created to replace it. The old block is reclaimed once the epoch counter has advanced to $e+2$, at which point we know that the new block will survive a crash. An old block can be *deleted* by replacing it with an "anti-block."

An operation that has updated a block in epoch e must abort and start over if it accesses any block that was created in epoch $e+1$; this ensures that the epoch boundary represents a consistent cut across the happens-before relationship. Epoch advance from e to $e+1$ must wait until (1) all blocks that were to be reclaimed in epoch $e-2$ have been reclaimed, and (2) all operations that began in epoch $e-1$ have completed and have persisted their updates.

Many details can be tuned for better performance. Buffered durable linearizability requires that updates to persistent data be written back and fenced by the end of the subsequent

epoch, but not necessarily before. To reduce the likelihood of cache misses induced by writing back (and, as a side effect, evicting) data that may still be useful, a thread can keep the addresses of its updates in a (transient) buffer for delayed write-back. To move the overhead of persistence off the critical path, buffers may be emptied by a background thread. To ensure nonblocking progress in application threads, updates to the epoch clock can also be confined to a background thread. Epoch length can be adjusted to trade run-time overhead against the amount of data that may be lost on a crash.

In the wake of a crash, Ralloc helps Montage iterate through all potentially in-use blocks in the heap, keeping those that are not from the two most recent epochs. The application then re-creates any needed transient structures.

3 Concrete examples

We have implemented several data structures in Montage, including a nonblocking queue, blocking and nonblocking mappings based on hash tables and trees, and general graphs with dynamic creation and deletion of vertices and edges. As a general rule, we avoid unbounded-length chains of pointers in persistent memory, since an update at the end of the chain could recursively necessitate new versions of all the predecessor blocks. Rather, we keep such chains in transient memory when needed, and arrange to rebuild them after a crash. In a queue, for example, each payload block contains a *sequence number* that indicates its place in line. In a graph, each edge is represented by a *relationship* block in persistent memory that contains the unique IDs of its endpoint vertices. Pointers from vertices to edges appear only in transient memory. Anecdotally, adapting a structure to Montage requires relatively modest programmer effort beyond the creation of the original concurrent object.

Figure 1 reports throughput for five persistent mappings: Dalí [5], synchronously logged and (on ≤ 20 threads) "full" (asynchronous) versions of Pronto [4], MOD [2], SOFT [6], and Montage. The Montage implementation uses a background thread to perform writes-back and advance the epoch every 100 ms. All implementations use a lock-based hash table as index. For comparison, we also measure a transient hash table (with data in DRAM or in NVM) and Montage without persistence (T). Only SOFT outperforms Montage, with the significant disadvantage of being limited by the size of DRAM. We interpret these results as a strong endorsement of buffered durable linearizability, and of Montage in particular.

References

1 W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott. Understanding and optimizing persistent memory allocation. In *19th Intl. Symp. on Memory Mgmt.*, June 2020.

2 S. Haria, M. D. Hill, and M. M. Swift. MOD: Minimally ordered durable datastructures for persistent memory. In *25th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, pages 775–788, March 2020.

3 J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Intl. Symp. on Dist. Comp.*, pages 313–327, September 2016.

4 A. Memaripour, J. Izraelevitz, and S. Swanson. Pronto: Easy and fast persistence for volatile data structures. In *25th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Sys.*, pages 789–806, March 2020.

5 F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott. Dalí: A periodically persistent hash map. In *Intl. Symp. on Dist. Comp.*, pages 37:1–37:16, October 2017.

6 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.