Building Fast Recoverable Persistent Data Structures

Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott
{hwen5,wcai6,mdu5,ljenkin4,bvalpey,scott}@cs.rochester.edu

1. Introduction

Beginning with Mnemosyne [8] and NV-Heaps [2] a decade ago, and continuing with such recent offerings as Pronto [6], many systems have been devised to support failure-atomic operations on data structures located in byte-addressable nonvolatile memory (NVM). All of these systems, to the best of our knowledge, share the property that when an operation completes and allows the caller to continue, the results are guaranteed to survive a crash. This property implies that, before returning, each operation must explicitly write back its stores and wait dozens or (depending on the hardware) even hundreds of cycles for an acknowledgment from the memory controller.

Figure 1 plots the throughput of several implementations of a large concurrent hash table, running on a machine equipped with Intel Optane NVM. Mnemosyne, the original transactional system for persistent memory, peaks at about 230 thousand operations per second. A conventional, transient table in DRAM can sustain more than 50 million operations per second—a difference of more than two orders of magnitude. Pronto is about 6× faster than Mnemosyne, but still more than 40× slower than the DRAM table. We aim to bridge that gap.

We propose *Montage*, a general system for building fast recoverable data structures. It embodies two key insights: ***First***, an operation need not persist everything it modifies; rather, it can persist just enough information to allow the data structure to be recovered after a crash. ***Second***, an operation need not persist before returning, so long as post-recovery system state is guaranteed to reflect a consistent prefix of pre-crash execution.

The first insight appears, for specific data structures, in several prior projects like MOD [3] and SOFT [10]. From a certain perspective, it might also be said to appear in Pronto. We explain how to embed it in a general-purpose system, applicable to arbitrary structures.

The second insight embodies the notion of *buffered durable linearizability*, a relaxed correctness criterion suggested (without concrete examples) by Izraelevitz et al. [4]. It also reflects long-standing practice in the HPC community, where periodic checkpoints allow an application to bound the amount of work that may be lost on a crash. The Dalí hash map [7], included in Figure 1, uses *periodic persistence* (buffering) to achieve more than twice the throughput of Pronto, but in a very data-structure–specific way. Montage again provides a general-purpose solution, applicable to arbitrary structures. As shown in Figure 1, our Montage hash map sustains well over 20 million operations per second on a read-heavy workload—

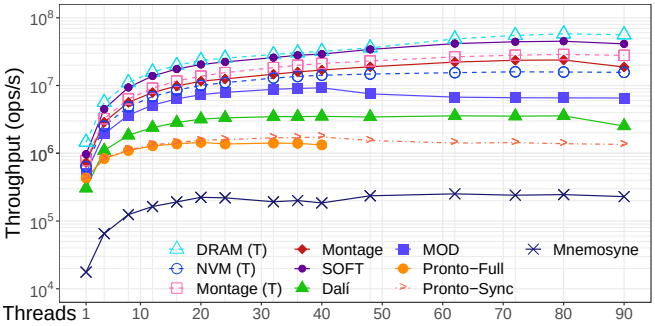

Figure 1: Hash table performance on a 2-socket, 80-thread machine (90% lookups). Note the log scale of the Y-axis.

7× faster than Dalí, 17× faster than Pronto, and within a factor of 3 of the DRAM table's throughput. This is close to the best one could hope for: Optane read bandwidth is about 3× that of DRAM.

Montage is designed to preserve the *abstract state* of a 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. Pre-crash execution employs a slow-running *epoch clock*. It ensures that no operation appears to span an epoch boundary, and these boundaries represent a consistent cut across the linearization order of the object. 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. Following the observation of Nawab et al. [7], Montage discards e and $e-1$ if crashes in e to allow operations in those epochs to overlap in time, avoiding the need for quiescence. If an application needs to be sure that a given operation has persisted (e.g., before confirming this to a remote client via network communication), it can invoke a sync() operation, which advances the epoch by 2.

2. Overview

Our implementation of Montage is built on top of Ralloc [1], a lock-free allocator for persistent memory. The Montage layer is also lock-free during normal operation, except for epoch advances: if the data structure itself is nonblocking, a stalled thread will not impede operations of its peers, but it can indefinitely prevent those operations from persisting. (We are currently developing mechanisms to make the progress of persistence nonblocking as well.)

A typical Montage structure consists of a collection of *payload* blocks in nonvolatile memory (NVM), together with a (much smaller) index or other supporting structure in DRAM. The global epoch clock is also kept in persistent memory. Each

payload block indicates the epoch in which it was created. 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 that was created in epoch $d < e$ is "modified" by replacing it with a new block. 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 *semantically deleted* by replacing it with an "anti-block."

An operation that is updating a block in epoch e must abort and start over if it encounters a dependence upon 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. We have generally obtained the best results by delaying all explicit writes-back until the end of the epoch, and performing both writes-back and memory reclamation in a dedicated background thread. Performance remains essentially constant across epoch lengths ranging from tens of microseconds to hundreds of milliseconds.

We have implemented several data structures in Montage, including queues, mappings based on hash tables and trees, and general graphs with dynamic creation and deletion of vertices and edges. 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.

Anecdotally, adapting a structure to Montage requires relatively modest programmer effort beyond the creation of the original concurrent object. Our C++ API provides operations to begin and end a failure-atomic operation, to allocate and deallocate payloads using Ralloc, and to get and set the fields of payload objects. The programmer is responsible for ensuring that each data structure operation linearizes in the epoch in which its payloads were created. This is trivial with locks. For nonblocking structures, we provide a special compare-and-swap operation (based on the double-compare-single-swap of Harris et al. [5]) that succeeds only if it can do so in the epoch of the invoking operation.

3. Experimental Results

Montage's performance generally exceeds that of prior general-purpose systems by large multiplicative factors, and equals or exceeds that of custom-designed persistent structures as well. In the hash table results of Figure 1, only SOFT [10] provides both persistence and better performance. Unfortunately, SOFT is unable (at least as currently realized) to exploit the high capacity of NVM—a copy of the entire structure must always reside in DRAM. Moreover, SOFT, like Dalí, is not a general purpose system, but a hash table that represents a set or mapping only. (And even as a mapping, it lacks an atomic *replace* operation for existing keys.)

In the full-length paper on ArXiv [9] we describe Montage in more detail; argue its correctness; explore its sensitivity to various design decisions, workload characteristics, and thread and memory placement; compare its performance to that of prior art, including both special-purpose data structures and general-purpose systems; document its recovery time; and validate our hash table results on a full-scale rebuild of memcached.

4. Conclusion

Montage is, to the best of our knowledge, the first general-purpose system for buffered (periodically persistent) data structures in nonvolatile memory, and the first to facilitate persisting only essential data, rebuilding the rest on recovery. In comparison to systems that are (strictly) durably linearizable, Montage moves write-back and fencing off the critical path of the application, allowing it to dramatically outperform existing special- and general-purpose systems for persistence. Montage combines this performance with very low conceptual complexity, paving the way for future work that would embed persistence in the programming language.

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 Management (ISMM)*, June 2020. Poster/brief announcement presented at the *25th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, CA, Feb. 2020.

[2] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, Newport Beach, CA, 2011.

[3] S. Haria, M. D. Hill, and M. M. Swift. MOD: Minimally ordered durable datastructures for persistent memory. In *25th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 775–788, Mar. 2020.

[4] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Intl. Symp. on Distributed Computing (DISC)*, pages 313–327, Paris, France, Sep. 2016.

[5] H. T. L., F. Keir, and P. I. A. A practical multi-word compare-and-swap operation. In *16th Intl. Symp. on Distributed Computing (DISC)*, pages 265–279, Toulouse, France, Oct. 2002.

[6] A. Memaripour, J. Izraelevitz, and S. Swanson. Pronto: Easy and fast persistence for volatile data structures. In *25th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 789–806, Mar. 2020.

[7] F. Nawab, J. Izraelevitz, T. Kelly, C. B. M. III, D. R. Chakrabarti, and M. L. Scott. Dalí: A periodically persistent hash map. In *Intl. Symp. on Distributed Computing (DISC)*, pages 37:1–37:16, Vienna, Austria, Oct. 2017.

[8] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, Newport Beach, CA, Mar. 2011.

[9] H. Wen, W. Cai, M. Du, L. Jenkins, B. Valpey, and M. L. Scott. Montage: A general system for buffered durably linearizable data structures. 2020. arXiv preprint arXiv:2009.13701.

[10] Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank. Efficient lock-free durable sets. *Proc. of the ACM on Programming Languages*, 3(OOPSLA):128:1–128:26, Oct. 2019.