Fast Nonblocking Persistence for Concurrent Data Structures

Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du,
Rafaello Sanna, Shreif Abdallah, and Michael L. Scott
{wcai6,hwen5,vmaksimo,mdu5,rsanna,selsaid,scott}@cs.rochester.edu

1. Introduction

Montage [6] is a general purpose system for building fast recoverable data structures located in byte-addressable nonvolatile memory (NVM). Montage employs two key techniques to minimize persistence overhead. First, it persists only the *abstract state* of a concurrent object. In a mapping, for example, only key-value pairs may be persisted, while the index (tree, hash table, skip list) can remain in transient DRAM. Second, it embraces the *buffered variant* of durable linearizability [4], which allows the latency of flushes and fences to be amortized over multiple operations.

Montage employs a slow-running clock that divides execution into *epochs*, and a background thread as the epoch advancer to periodically persist updates in the old epoch and increment the clock. To guarantee that post-recovery system state reflects a consistent prefix of pre-crash execution, updates in an old epoch must linearize before those in a new epoch, and an epoch advance to e waits for all updates in $e-2$ to complete. If a crash happens in epoch e, Montage recovers the structure to the end of epoch $e-2$. Optionally, the user may invoke a `sync` operation to request two epoch advances, ensuring the persistence of a given operation.

Built on top of Ralloc [1], a lock-free allocator for persistent memory, Montage guarantees lock-freedom during normal operation, but the epoch advances are blocking: if the structure itself is lock-free, it may still make forward progress despite any stalled thread, but the advancer must wait for any stalled thread to leave the epoch before proceeding. This potentially indefinite waiting precludes a bound on the number of updates that may be lost on a crash or the time required to complete a `sync`. Because `sync` is implemented in a background thread that must interact with all other threads, latency increases dramatically with thread count, as shown in Figure 1.

To address these progress and performance issues, we have developed *nbMontage* [2], a nonblocking variant of the original Montage tailored for nonblocking data structures, and with a fast, wait-free `sync`. The implementation is based on the observation that to ensure consistency on a crash, each operation must linearize in the epoch in which it created data. This can be ensured if the linearization point is a compare-and-swap (CAS) instruction that we can replace with a software *double-compare-single-swap* (DCSS [3]) that verifies both the expected value in memory *and* the current epoch. By making the DCSS visible to other threads, we can arrange for `sync` to *abort* slow operations instead of waiting for them. The aborted operations then re-start in the newer epoch.

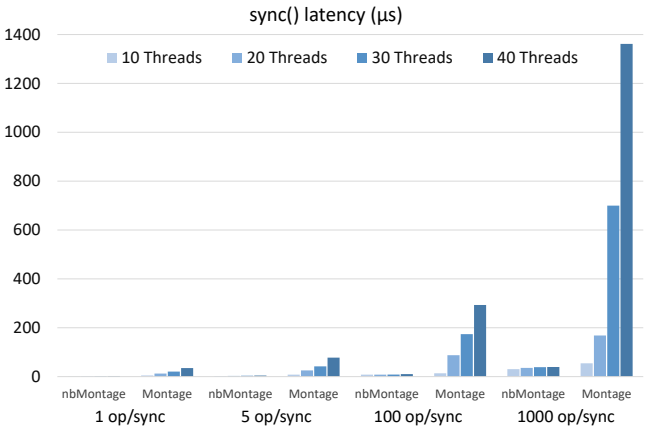

Figure 1: Average latency of `sync` on hash tables.

Unfortunately, the waiting mechanism is deeply embedded in the original Montage implementation. In the following section, we summarize four significant changes required to obtain a wait-free, responsive `sync` while still preserving Montage's negligible persistence overhead. Full details can be found in our paper at DISC '21 [2].

2. nbMontage

As in the original Montage, a typical nbMontage structure consists of a collection of nonvolatile *payload* blocks that capture abstract state, together with a much smaller index or other auxiliary structure in DRAM. The global epoch clock is also kept persistently in NVM. Each payload block is labeled with the epoch of the operation that created it. A write operation (in epoch $e-2$, for example) replaces an old payload block (if any) with a new block. The old one is reclaimed after the epoch clock has incremented to e, at which point we know that the new block will survive a crash and the update has persisted. Similarly, a `delete` replaces an old block with a special "anti-block" that can itself be reclaimed in two epochs. Major changes to the original system are as follows.

First, we deprecate the `begin_op` and `end_op` API calls, which explicitly annotated failure-atomic sections, and introduce a new, intuitive `lin_CAS` API, to simply replace linearizing CAS of nonblocking operations, along with `pnew` and `pdetach` functions that allocate payload blocks and anti-blocks of pending operations prior to their linearization points. At each `lin_CAS`, nbMontage annotates blocks allocated in the current epoch, performs an epoch-verifying DCSS, and automatically retries if the DCSS fails due to epoch change. If an operation stalls after linearizing the DCSS but before

cleaning up its metadata, an epoch advancer or competing thread will persist already staged updates.

Second, recovery must distinguish between payloads that were created by committed and failed updates. To support this, we persist DCSS descriptors and use thread ID and a monotonic serial number to associate a unique ID with every update. Each thread has a single, reusable descriptor; still-extant payloads with earlier serial numbers can be assumed to have committed; those corresponding to the serial number of an aborted descriptor can be discarded, even if they belong to an otherwise persisted epoch.

Third, to tolerate doomed threads in an arbitrary number of old epochs, we redesign the set containers used to buffer to-be-persisted (TBP) payloads and to-be-freed (TBF) payloads in each epoch: TBP containers are implemented as wait-free single-producer-multiple-consumer FIFO circular buffers, intentionally allowing duplicated pops for better performance, as duplicated flushes are benign; TBF containers, on the other hand, can be handled lazily by the freeing thread.

Fourth, we use a locality-aware variant of Liu et al.'s *Mindicator* [5] structure to implement a fast, decentralized, collaborative `sync`. The structure is a fixed-size, wait-free balanced tree, in which each leaf represents a thread and indicates the epoch of its current operation. Each interior node indicates the minimum epoch in its subtree. Any thread that invokes `sync` can find the epochs and operations that need to be persisted by scanning up and down from the thread's own leaf. NUMA locality is reflected in the structure of the tree, leading to locality-aware search and helping, and reducing `sync` latency to a few microseconds. As shown in Figure 1, the improvement with respect to the original Montage can be two full orders of magnitude.

In general, nbMontage supports any nonblocking data structures whose write operations linearize at a compare-and-swap (CAS) and that know, immediately after the CAS returns, whether it forms the linearization point. Most nonblocking structures in the literature appear to meet this requirement. We have adapted numerous examples, including queues, skiplists, hash tables, and binary search trees. With the new API, the adaptation is straightforward and "mostly mechanical."

3. Experimental Results

Like that of the original Montage, nbMontage's performance generally exceeds that of both prior general-purpose systems and custom-designed persistent structures. Figure 2 shows the throughput of million-element hash tables with an 18–1–1 ratio of lookups, inserts, and removes. Only SOFT [7] offers both persistence and higher performance. Because it keeps a full copy of its data in DRAM, however, SOFT is unable to exploit the high capacity of NVM. It also employs optimizations that preclude provision of an atomic `replace` operation for existing keys.

Our experiments employ an epoch length of 10 ms by default, but throughput remains remarkably high and stable even

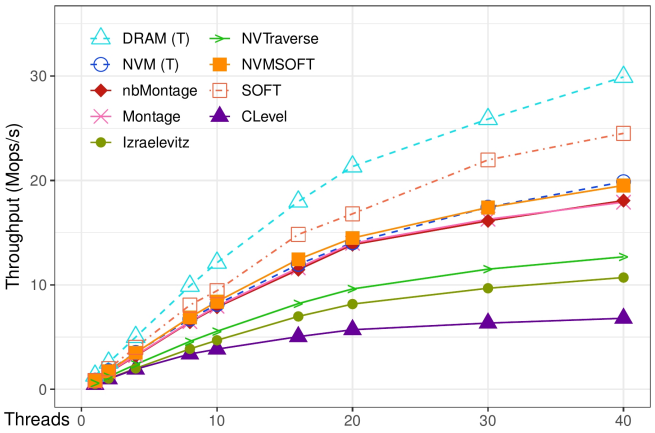

Figure 2: Hash tables on a 40-thread CPU (90% lookups).

when the epoch length is as short as 10 µs. More experiments and example code for a nonblocking hash table can be found in the full-length paper [2].

4. Conclusion

To the best of our knowledge, nbMontage is the first general-purpose system to combine buffered durable linearizability and nonblocking progress of the persistence frontier. Nonblocking persistence allows nbMontage to provide a fast wait-free `sync` routine and to strictly bound the work that may be lost on a crash. Experience with a variety of nonblocking data structures confirms that they are easy to port to nbMontage, and perform extremely well. Given that programmers have long been accustomed to `sync`-ing their updates to file systems and databases, a system with the performance and formal guarantees of nbMontage appears to be of significant practical utility. The code of both Montage and nbMontage is publicly available at github.com/urcs-sync/montage.

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.

[2] W. Cai, H. Wen, V. Maksimovski, M. Du, R. Sanna, S. Abdallah, and M. L. Scott. Fast Nonblocking Persistence for Concurrent Data Structures. In *35th Intl. Symp. on Distributed Computing (DISC 2021)*, pages 14:1–14:20, 2021. Extended version arXiv:2105.09508.

[3] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *16th Intl. Symp. on Distributed Computing (DISC)*, pages 265–279, Toulouse, France, Oct. 2002.

[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] Y. Liu, V. Luchangco, and M. Spear. Mindicators: A scalable approach to quiescence. In *IEEE 33rd Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 206–215, Philadelphia, PA, 2013.

[6] H. Wen, W. Cai, M. Du, L. Jenkins, B. Valpey, and M. L. Scott. A fast, general system for buffered persistent data structures. In *50th Intl. Conf. on Parallel Processing (ICPP)*, Aug. 2021.

[7] 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.