Fast Nonblocking Persistence for Concurrent Data Structures

Wentao Cai[1]
University of Rochester, NY, USA

Haosen Wen[1]
University of Rochester, NY, USA

Vladimir Maksimovski
University of Rochester, NY, USA

Mingzhe Du
University of Rochester, NY, USA

Rafaello Sanna
University of Rochester, NY, USA

Shreif Abdallah
University of Rochester, NY, USA

Michael L. Scott
University of Rochester, NY, USA

Abstract

We present a fully lock-free variant of our recent Montage system for persistent data structures. The variant, nbMontage, adds persistence to almost any nonblocking concurrent structure without introducing significant overhead or blocking of any kind. Like its predecessor, nbMontage is *buffered durably linearizable*: it guarantees that the state recovered in the wake of a crash will represent a consistent prefix of pre-crash execution. Unlike its predecessor, nbMontage ensures wait-free progress of the persistence frontier, thereby bounding the number of recent updates that may be lost on a crash, and allowing a thread to force an update of the frontier (i.e., to perform a `sync` operation) without the risk of blocking. As an extra benefit, the helping mechanism employed by our wait-free `sync` significantly reduces its latency.

Performance results for nonblocking queues, skip lists, trees, and hash tables rival custom data structures in the literature – dramatically faster than achieved with prior general-purpose systems, and generally within 50% of equivalent non-persistent structures placed in DRAM.

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

Keywords and phrases Persistent Memory, Nonblocking Progress, Buffered Durable Linearizability

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.14

Related Version *Full Version*: `https://arxiv.org/abs/2105.09508`

Supplementary Material *Software (Source Code)*: `https://github.com/urcs-sync/Montage` archived at `swh:1:dir:421cef8ca9c97f07099163c791aa8316c60fda86`

Funding This work was supported in part by NSF grants CCF-1717712, CNS-1900803, CNS-1955498, and by a Google Faculty Research award.

1 Introduction

With the advent of dense, inexpensive nonvolatile memory (NVM), it is now feasible to retain pointer-based, in-memory data structures across program runs and even hardware reboots. So long as caches remain transient, however, programs must be instrumented with write-back and fence instructions to ensure that such structures remain consistent in the wake of a crash. Minimizing the cost of this instrumentation remains a significant challenge.

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

Section 2 of this paper surveys both bespoke persistent data structures and general-purpose systems. Most of these structures and systems ensure that execution is *durably linearizable* [28]: not only will every operation linearize (appear to occur atomically) sometime between its call and return; each will also *persist* (reach a state in which it will survive a crash) before returning, and the persistence order will match the linearization order. To ensure this agreement between linearization and persistence, most existing work relies on locks to update the data structure and an undo or redo log together, atomically.

Unfortunately, strict durable linearizability forces expensive write-back and fence instructions onto the critical path of every operation. A few data structures – e.g., the Dalí hash table of Nawab et al. [43] and the InCLL MassTree of Cohen et al. [10] – reduce the cost of instrumentation by supporting only a relaxed, *buffered* variant [28] of durable linearizability. Nawab et al. dub the implementation technique *periodic persistence*. In the wake of post-crash recovery, a buffered durably linearizable structure is guaranteed to reflect some prefix of the pre-crash linearization order – but with the possibility that recent updates may be lost.

Independently, some data structures – notably, the B-tree variants of Yang et al. [54], Oukid et al. [44], and Chen et al. [7], and the hash set of Zuriel et al. [55] – reduce the overhead of persistence by writing back and fencing only the data required to rebuild a semantically equivalent structure during post-crash recovery. A mapping, for example, need only persist a pile of key-value pairs. Index structures, which serve only to increase performance, can be kept in faster, volatile memory.

Inspired by Dalí and the InCLL MassTree, our group introduced the first *general purpose* system [53] for buffered durably linearizable data structures. Like the sets of the previous paragraph, *Montage* facilitates persisting only semantically essential *payloads*. Specifically, it employs a global *epoch clock*. It tracks the semantically essential updates performed in each operation and ensures that (1) no operation appears to span an epoch boundary, and (2) no essential update fails to persist for two consecutive epochs. If a crash occurs in epoch e, Montage recovers the abstract state of the structure from the end of epoch $e-2$.

In return for allowing a bit of work to be lost on a crash, and for being willing to rebuild, during recovery, any portion of a structure needed only for good performance, users of Montage can expect to achieve performance limited primarily by the read and write latency of the underlying memory – not by instrumentation overhead. When a strict persistence guarantee is required (e.g., before printing a completion notice on the screen or responding to a client over the network), Montage allows an application thread to invoke an explicit `sync` operation – just as it would for data in any conventional file or database system.

Unfortunately, like most general-purpose persistence systems, Montage relies on locks at critical points in the code. All of the reported performance results are for lock-based data structures. More significantly, while the system can support nonblocking structures, progress of the persistence frontier is fundamentally blocking. Specifically, before advancing the epoch from e to $e+1$, the system waits for all operations active in epoch $e-1$ to complete. If one of those operations is stalled (e.g., due to preemption), epoch advance can be indefinitely delayed. This fact implies that the `sync` operation cannot be nonblocking. It also implies that the size of the history suffix that may be lost on a crash cannot be bounded *a priori*.

While blocking is acceptable in many circumstances, nonblocking structures have compelling advantages. In the original paper on durable linearizability, Izraelevitz et al. [28] presented a mechanical transform that will turn any linearizable nonblocking concurrent data structure into an equivalent persistent structure. Unfortunately, this transform is quite expensive, especially for highly concurrent programs: it litters the code with write-back and fence instructions. More efficient strategies for several specific data structures, including queues [19] and hash tables [8, 55], have also been developed by hand.

Friedman et al. [18] observed that many nonblocking operations begin with a read-only "traversal" phase in which instrumentation can, with care, be safely elided; applying this observation to the transform of Izraelevitz et al. leads to substantially better performance in many cases, but the coding process is no longer entirely mechanical. Ramalhete et al. [46] and Beadle et al. [1] present nonblocking persistent software transactional memory (STM) systems, but both have fundamental serial bottlenecks that limit scalability.

In this paper, we extend Montage to produce the first *general-purpose, high-performance system for nonblocking periodic persistence*. Our *nbMontage* allows programmers, in a straightforward way, to convert most wait-free or lock-free linearizable concurrent data structures into fast, equivalent structures that are lock-free and buffered durably linearizable. (Obstruction-free structures can also be converted; they remain obstruction free.) Like its predecessor, nbMontage requires every nonblocking update operation to linearize at a compare-and-swap (CAS) instruction that is guaranteed, prior to its execution, to constitute the linearization point if it succeeds. (Read-only operations may linearize at a `load`.) Most nonblocking structures in the literature meet these requirements.

Unlike its predecessor, nbMontage provides a wait-free `sync`. Where the original system required all operations in epoch $e-1$ to complete before the epoch could advance to $e+1$, nbMontage allows pending operations to remain in limbo: in the wake of a crash or of continued crash-free execution, an operation still pending at the end of $e-1$ may or may not be seen, sometime later, to have linearized in that epoch. In addition, where the original Montage required operations to accommodate "spurious" CAS failures caused by epoch advance, nbMontage retries such CASes internally, without compromising lock freedom. These changes were highly nontrivial: they required new mechanisms to register (announce) pending updates; distinguish, in recovery, between registered and linearized updates; indefinitely delay (and reason about) the point at which an update is known to have linearized; and avoid work at `sync` time for threads that have nothing to persist.

We have applied nbMontage to Michael & Scott's queue [41], Natarajan & Mittal's binary search tree [42], the rotating skip list of Dick et al. [15], Michael's hash table [39], and Shalev & Shavit's extensible hash table [48]. All conversions were straightforward – typically less than 30 lines of code. By persisting only essential data and avoiding writes-back and fences on the critical path, nbMontage structures often approach or outperform not only their equivalents in the original (blocking) Montage but also their transient equivalents when data is placed (without any algorithmic changes) in slower, nonvolatile memory.

Summarizing contributions:

1. We introduce nbMontage, the first general system for nonblocking periodic persistence.
2. We tailor the nbMontage API to nonblocking data structures. With this new API, conversion of existing nonblocking structures for persistence is straightforward.
3. We argue that nbMontage provides buffered durable linearizability and wait-free `sync` for compatible data structures, while still preserving safety and liveness.
4. We compare the performance of nbMontage structures both to their original, transient versions and to hand-crafted alternatives from the literature, running on a recent Intel server with Optane NVM. Our results confirm exceptional throughput, responsive `sync`, negligible overhead relative to the original Montage, and reasonable recovery latency.

2 Related Work

The past decade has seen extensive work on persistent data structures, much of it focused on B-tree indices for file systems and databases [3, 6, 7, 26, 29, 31, 44, 50, 54]. Other work has targeted queues [19], RB trees [52], radix trees [30], and hash tables [8, 43, 47, 55].

In recent years, *durable linearizability* has emerged as the standard correctness criterion for such structures [19, 28, 37, 55]. This criterion builds on the familiar notion of linearizability [24] for concurrent (non-persistent) structures. A structure is said to be linearizable if, whenever threads perform operations concurrently, the effect is as if each operation took place, atomically, at some point between its call and return, yielding a history consistent with the sequential semantics of the abstraction represented by the structure. A persistent structure is said to be *durably linearizable* if (1) it is linearizable during crash-free execution, (2) each operation persists (reaches a state that will survive a crash) at some point between its call and return, and (3) the order of persists matches the linearization order.

In addition to custom data structures, several groups have developed general-purpose systems to ensure the failure atomicity of lock-based critical sections [4, 25, 27, 33] or speculative transactions [1, 5, 9, 11, 13, 20–22, 32, 38, 45, 46, 49, 51]. Like the bespoke structures mentioned above, all of these systems are durably linearizable – they ensure that an operation has persisted before returning to the calling thread.

As noted in Section 1, nonblocking persistent structures can achieve failure atomicity without the need for logging, since every reachable concrete state corresponds to a well-defined abstract state. At the same time, while a lock-based operation can easily arrange to linearize and persist in the same order (simply by holding onto the locks needed by any successor operation), a nonblocking operation becomes visible to other threads the moment it linearizes. As a result, those other threads must generally take care to ensure that anything they read has persisted before they act upon it. Writing back and fencing read locations is a major source of overhead in the mechanical transform of Izraelevitz et al. [28]. Friedman et al. [18] avoid this overhead during the initial "traversal" phase of certain nonblocking algorithms. David et al. [14] avoid redundant writes-back and fences in linked data structures by *marking* each updated pointer in one of its low-order bits. A thread that persists the pointer can use a CAS to clear the bit, informing other threads that they no longer need to do so.

In both blocking and nonblocking structures, the overhead of persistence can be reduced by observing that not all data are semantically meaningful. In any implementation of a set or mapping, for example, the items or key-value pairs must be persistent, but the index structure can (with some effort) be *rebuilt* during recovery. Several groups have designed persistent B-trees, lists, or hash tables based on this observation [7, 35, 44, 54].

Ultimately, however, any data structure or general-purpose system meeting the strict definition of durable linearizability will inevitably incur the overhead of at least one persistence fence in every operation [12]. For highly concurrent persistent structures, this overhead can easily double the latency of every operation. Similar observations, of course, have applied to I/O operations since the dawn of electronic computing, and are routinely addressed by completing I/O in the background. For data structures held in NVM, *periodic persistence* [43] provides an analogous "backgrounding" strategy, allowing a structure to meet the more relaxed requirements of *buffered* durable linearizability – specifically, the state recovered after a crash is guaranteed to represent a prefix of the linearization order of pre-crash execution.

Nawab et al. [43] present a lock-based hash table, Dalí, that performs updates only by pre-pending to per-bucket lists, thereby creating a revision history (deletions are effected by inserting "anti-nodes"). A clever "pointer-rotation" strategy records, for each bucket, the head nodes of the list for each of the past few values of a global *epoch clock*. At the end of each coarse-grain epoch, the entire cache is flushed. In the wake of a crash, threads ignore nodes prepended to hash bucket lists during the two most recent epochs. No other writes-back or fences are required. Cohen et al. [10] also flush the cache at global epoch boundaries, but employ a clever system of *in-cache-line-logging* (InCLL) to retain the epoch

number and beginning-of-epoch value for every modified leaf-level pointer in a variant of the Masstree structure [36]. In the wake of a crash, threads use the beginning-of-epoch value for any pointer that was modified in the epoch of the crash.

Both Dalí and InCLL employ techniques that might be applicable in principle to other data structures. To the best of our knowledge, however, Montage [53] is the only existing general-purpose system for buffered durable linearizable structures. It also has the advantage of persisting only semantically essential data. As presented, unfortunately, it provides only limited support for nonblocking data structures, and its attempts to advance the epoch clock can be arbitrarily delayed by stalled worker threads. Our nbMontage resolves these limitations, allowing us to provide a wait-free `sync` operation and to bound the amount of work that may be lost on a crash. It also provides a substantially simpler API.

3 System Design

3.1 The Original Montage

Semantically essential data in Montage resides in *payload* blocks in NVM. Other data may reside in transient memory. The original system API [53] is designed primarily for lock-based data structures, but also includes support for nonblocking operations (with blocking advance of the persistence frontier). The typical operation is bracketed by calls to `begin_op` and `end_op`. In between, reads and writes of payloads use special *accessor* (`get` and `set`) methods.

Internally, Montage maintains a slow-ticking *epoch clock*. In the wake of a crash in epoch e, the Montage recovery procedure identifies all and only the payloads in existence at the end of epoch $e-2$. It provides these, through a parallel iterator, to the restarted application, which can then rebuild any needed transient structures. Accessor methods allow payloads that were created in the current epoch to be modified in place, but they introduce significant complexity to the programming model.

Payloads are created and deleted with `pnew` and `pdelete`. These routines are built on a modified version of our Ralloc [2] persistent allocator. In the original Ralloc, a tracing garbage collector was used in the wake of a crash to identify and reclaim free blocks. In the Montage version, epoch *tags* in payloads are used to identify all and only those blocks created and not subsequently deleted at least two epochs in the past. To allow a payload to survive if a crash happens less than two epochs after a deletion, deletions are implemented by creating *anti-nodes*. These are automatically reclaimed, along with their corresponding payloads, after two epochs have passed.

To support nonblocking operations, the original Montage provides a `CAS_verify` routine that succeeds only if it can do so in the same epoch as the preceding `begin_op`. If `CAS_verify` fails, the operation will typically start over; before doing so, it should call `abort_op` instead of `end_op`, allowing the system to clean up without persisting the operation's updates.

Regardless of persistence, nodes removed from a nonblocking structure typically require some sort of *safe memory reclamation* (SMR) – e.g., epoch-based reclamation [17] or hazard pointers [40] – to delay the reuse of memory until one can be sure that no thread still holds a transient reference. In support of SMR, the original Montage provides a `pretire` routine that creates an anti-node to mark a payload as semantically deleted, and no new operation can reference this payload. In the absence of crashes, Montage will automatically reclaim the payload and its anti-node 2–3 epochs after SMR calls `pdelete`. In the event of a crash, however, if two epochs have elapsed since `pretire`, the existence of the anti-node allows the Montage recovery procedure to avoid a memory leak by reclaiming the retired payload even in the absence of `pdelete`. This is safe since the epoch of the `pretire` is persisted, and all

```
class PBlk; // Base class of all Payload classes
class Recoverable { // Base class of all persistent objects
  template <class payload_type> payload_type* pnew(...); // Create a payload block
  void pdelete(PBlk*); // Delete a payload; should be called only when safe, e.g., by SMR.
  void pdetach(PBlk*); // Mark payload for retirement if operation succeeds
  void sync(); // Persist all operations that happened before this call
  vector<PBlk*>* recover(); // Recover and return all recovered payloads
  void abort_op(); // Optional routine to abandon current operation
};
template <class T> class CASObj { // Atomic CAS object that provides load and CAS
  /* Epoch-verifying linearization method: */
  bool lin_CAS(T expected, T desired) {
    begin_op(); // write back or delete old payloads as necessary; tag new ones
    while (1) { // iterations can be limited for liveness
      ... // main body of DCSS
      switch (DCSS_status) {
        case COMMITTED: end_op(); return true; // clean up metadata
        case FAILED: abort_op(); return false; // untag payloads and clear retirements; clean up metadata
        case EPOCH_ADVANCED: reset_op(); // update and reuse payloads and retirements
      }
    }
  /* Non-verifying atomic methods: */
  T load(); void store(T desired); bool CAS(T expected, T desired);
};
```

Figure 1 C++ API of nbMontage, largely inherited from the original Montage.

operations with references to this payload are gone after the crash. Significantly, since a still-active operation will (in the original Montage) prevent the epoch from advancing far enough to persist anything in its epoch, `pretire` can safely be called after the operation has linearized, so long as it has not yet called `end_op`.

When a program needs to ensure that operation(s) have persisted (e.g., before printing a confirmation on the screen or responding to a client over the network), Montage allows it to invoke an explicit `sync`. The implementation simply advances the epoch from e to $e+2$ (waiting for operations in epochs $e-1$ and e to complete). The two-epoch convention avoids the need for quiescence [43]: a thread can advance the epoch from e to $e+1$ while other threads are still completing operations that will linearize in e.

The key to buffered durable linearizability is to ensure that every operation that updates payloads linearizes in the epoch with which those payloads are tagged. Each epoch boundary then captures a prefix of the data structure's linearization order. Maintaining this consistency is straightforward for lock-based operations. In the nonblocking case, `CAS_verify` uses a variant of the double-compare-single-swap (DCSS) construction of Harris al. [23] (see App. B in Full Version for complete pseudocode) to confirm the expected epoch and perform a conditional update, atomically. Unfortunately, the fact that an epoch advance from e to $e+1$ must wait for operations in $e-1$ means that even if a data structure remains nonblocking during crash-free execution, the progress of persistence itself is fundamentally blocking. This in turn implies that calls to `sync` are blocking, and that it is not possible, *a priori*, to bound the amount of work that may be lost on a crash.

Note, however, that since `CAS_verify` will succeed only in the expected epoch, any nonblocking operation that lags behind an epoch advance is doomed to fail and retry in a subsequent epoch. There is no need to wait for it to resume, explicitly fail, and unregister from the old epoch in `abort_op`. Unfortunately, the waiting mechanism is deeply embedded in the original Montage implementation – e.g., in the implementation of `pretire`, as noted above. Overall, there are four nontrivial issues that must be addressed to build nbMontage:

(Sec. 3.3) Every operation must register its pending updates (both new and to-be-deleted payloads) before reaching its linearization point, so that an epoch advance can help it to persist even if it stalls immediately after the linearization point.

(Sec. 3.4) The recovery procedure must be able to distinguish an epoch's "real" payloads and anti-nodes from those that may have been registered for an operation that failed due to a CAS conflict or epoch advance.

(Sec. 3.5) The buffering containers that record persistent blocks to be written back or deleted need a redesign, in order to accommodate an arbitrary number of epochs in which operations have not yet noticed that they are doomed to fail and retry in a new epoch.

(Sec. 3.6) An epoch advance must be able to find and persist any blocks (payloads or anti-nodes) that were created in the previous epoch but have not yet been written back and fenced. If `sync` is to be fast, this search must avoid iterating over all active threads.

3.2 nbMontage API

As shown in Figure 1, the nbMontage API reflects three major changes. First, because the epoch can now advance from e to $e+1$ even when an operation is still active in $e-1$, we must consider the possibility that a thread may remove a payload from the structure, linearize, and then stall. If a crash occurs two epochs later, we must ensure that the removed payload is deleted during post-crash recovery, to avoid a memory leak: post-linearization `pretire` no longer suffices. Our nbMontage therefore replaces `pretire` with a `pdetach` routine that must be used to register to-be-deleted payloads *prior* to linearization. As in the original Montage, deletion during crash-free execution is the responsibility of the application's SMR.

Second, again because of nonblocking epoch advance, nbMontage requires that payloads visible to more than one thread be treated as immutable. Updates always entail the creation of new payloads; `get` and `set` accessors are eliminated.

Third, in a dramatic simplification, nbMontage replaces the original `begin_op`, `end_op`, and `CAS_verify` with a new `lin_CAS` (linearizing CAS) routine. (The `abort_op` routine is also rolled into `lin_CAS`, but remains in the API so operations can call it explicitly if they choose, for logical reasons, to start over.) When `lin_CAS` is called, all payloads created by the calling thread since its last completed operation (and not subsequently deleted) will be tagged with the current epoch, e. All anti-nodes stemming from `pdetach` calls made since the last completed operation (and not corresponding to payloads created in that interval) will likewise be tagged with e. The `lin_CAS` will then attempt a DCSS and, if the current epoch is still e and the specified location contains the expected value, the operation will linearize, perform the internal cleanup previously associated with `end_op`, and return `true`. If the DCSS fails because of a conflicting update, `lin_CAS` will perform the internal cleanup associated with `abort_op` and return `false`. If the DCSS fails due to epoch advance, `lin_CAS` will update the operation's payloads and anti-nodes to the new epoch and retry. By ensuring, internally, that the epoch never advances unless some thread has completed an operation (App. C in Full Version), nbMontage can ensure that some thread makes progress in each iteration of the retry loop.

Programmers using nbMontage are expected to obey the following constraints:

1. Each nbMontage data structure R must be designed to be nonblocking and linearizable during crash-free execution when nbMontage is disabled. Specifically, R must be linearizable when (a) `pnew` and `pdelete` are ordinary `new` and `delete`; (b) `pdetach` and `sync` are no-ops; and (c) `CASObj` is `std::atomic` and `lin_CAS` is ordinary `CAS`.
2. Every update operation of R linearizes in a pre-identified `lin_CAS`– one that is guaranteed, before the call, to comprise the operation's linearization point if it succeeds. Any operation that conflicts with or depends upon this update must access the `lin_CAS`'s target location. Read-only operations may linearize at a `load`.

3. Once attached to a structure (made visible to other threads), a payload is immutable. Changes to a structure are made only by adding and removing payloads.
4. The semantics of each operation are fully determined by the set of payloads returned by `pnew` and/or passed to `pdetach` prior to `lin_CAS`.

Pseudocode for nbMontage's core classes and methods appears in Figure 2; these are discussed and referred to by pseudocode line numbers in the following subsections. Appendix A presents the changes required to port Maged Michael's lock-free hash table [39] to nbMontage.

3.3 Updates to Payloads

To allow the epoch clock to advance without blocking, nbMontage abandons in-place update of payloads. It interprets `pdetach` as requesting the creation of an *anti-node*. An anti-node shares an otherwise unique, hidden ID with the payload being detached. Newly created payloads and anti-nodes are buffered until the next `lin_CAS` in their thread. If the `lin_CAS` succeeds, the buffered nodes will be visible to epoch advance operations, and will persist even if the creating thread has stalled.

In the pseudocode of Figure 2, calls to `pnew` and `pdetach` are held in the `allocs` and `detaches` containers. Anti-nodes are created, and both payloads and anti-nodes are tagged, in `begin_op` (lines 69–76, called from within `lin_CAS`). If the `lin_CAS` fails due to conflict, `abort_op` resets `pnew`-ed payloads so they can be reused in the operation's next attempt (lines 91–92); it also withdraws `pdetach` requests, allowing the application to detach something different the next time around (lines 87–90). If attempts occur in a loop (as is common), the programmer may call `pnew` outside the loop and `pdetach` inside, as shown in Figure 7 (App. A). If an operation no longer needs its `pnew`-ed payloads (e.g., after a failed insertion), it may call `pdelete` to delete them; this automatically erases them from `allocs` (line 56). The internal `reset_op` routine serves to update and reuse both payloads and anti-nodes in anticipation of retrying a DCSS that fails due to epoch advance (lines 95–97).

3.4 CAS and Recovery

The implementation of `lin_CAS` employs an array of persistent *descriptors*, one per thread. These form the basis of the DCSS construction [23] (App. B in Full Version). Each descriptor records CAS parameters (the old value, new value, and CAS object); the epoch in which to linearize; and the status of the CAS itself (in progress, committed, or failed – lines 29–37). After a crash, the recovery procedure must be able to tell when a block in NVM (a payload or anti-node) appears to be old enough to persist, but corresponds to an operation that did not commit. Toward that end, each block contains a 64-bit field that encodes the thread ID and a monotonic serial number; together, these constitute a unique *operation ID* (lines 32 and 43). At the beginning of each operation attempt, `begin_op` updates the descriptor, incrementing its serial number (line 65). Previous uses of the descriptor with smaller serial numbers are regarded as having committed; blocks corresponding to those versions remain valid unless they are deleted or reinitialized (lines 67, 88, and 92). Deleting or reinitializing a persistent block resets its epoch to zero and registers it to be written back in the current epoch (lines 44–51). Registration ensures that resets persist, in `begin_op`, *before the next update to the descriptor* (lines 61–65). During an epoch advance from e to $e + 1$, the descriptors of operations in $e - 1$ are written back (at line 102) to ensure that their statuses reach NVM before the update of the global epoch clock.

Informally, an nbMontage payload is said to be *in use* if it has been created and not yet detached by linearized operations. Identifying such payloads precisely is made difficult by the existence of *pending* operations – those that have started but not yet completed, and

```
1  Struct CircBuffer
2      uint64 cap
3      atomic<uint64> pushed,popped
4      PBlk* blks[cap]
5      Function push(PBlk* item)
6          cpush=pushed.load()
7          cpop=popped.load()
8          if cpush≥cpop+cap then
9              clwb(blks[cpop%cap])
10             popped.CAS(cpop,cpop+1)
11         blks[cpush%cap]=item
12         pushed.store(cpush+1) // single producer
13     Function pop_all()
14         cpop=popped.load()
15         cpush=pushed.load()
16         if cpop==cpush then
17             return
18         foreach i from cpop to cpush do
19             break if i%cap reaches cpop%cap twice
20             clwb(blks[i%cap])
21         popped.CAS(cpop,cpush)
22 atomic<uint64> global_epoch
23 Mindicator mindi
24 thread_local uint64 e_curr,e_last
25 thread_local vector<PBlk*> allocs,detaches
26 thread_local int tid
27 int thd_cnt // number of threads
28 CircBuffer TBP[thd_cnt][4]
29 Struct Desc
30     uint64 old,new,epoch=0
31     uint64 type=DESC
32     uint64 tid_sn=0
       // 64 bits for ref to CASObj
       // 64 for cnt, with last 2 for status
33     atomic<uint128> r_c_s
34     Function reinit(uint64 e)
35         tid_sn++
36         r_c_s.store(0)
37         epoch=e
38 Struct PBlk
39     (void* vtable)
40     PBlk* anti=NULL
41     uint64 epoch=0
42     uint64 type={PAYLOAD,ANTI}
43     uint64 tid_sn=0,blk_uid=0
44     Function setup(uint64 t,Desc* desc,uint64 e)
45         type=t
46         epoch=desc?0:desc.epoch
47         tid_sn=desc?0:desc.tid_sn
48         TBP[tid][e%4].push(this)
49     Function destructor()
50         epoch=0
51         clwb(this)
52 Desc descs[thd_cnt]
53 vector<PBlk*> TBF[thd_cnt][4]
54 Function pdelete(PBlk* pblk)
55     e=global_epoch.load()
56     allocs.erase(pblk) // no-op if pblk not in allocs
57     TBF[tid][(e+1)%4].insert(pblk→anti)
58     TBF[tid][e%4].insert(pblk)
59 Function begin_op(bool reset=false)
60     e_curr=global_epoch.load()
61     if e_last<e_curr then
62         TBP[tid][e_last%4].pop_all()
63         clwb(descs[t])
64         update t's val to e_curr in mindi
65     descs[tid].reinit(e_curr)
66     for i from e_last-1 to min(e_last+1,e_curr-2) do
67         delete all items in TBF[tid][i%4]
68         sfence
69     foreach r in detaches do
70         if !reset then // default branch
71             allocate an anti-node anti for r
72             r→anti=anti
73             r→anti→blk_uid=r→blk_uid
74         r→anti→setup(ANTI,descs[tid],e_curr)
75     foreach p in allocs do
76         p→setup(PAYLOAD,descs[tid],e_curr)
77 Function end_op()
78     detaches.clear()
79     allocs.clear()
80     e_last=e_curr
81     e_curr=0
82 Function abort_op(bool reset=false)
83     if reset then // to reuse anti-nodes
84         foreach r in detaches do
85             r→anti→setup(ANTI,NULL,e_curr)
86     else // default branch:delete anti-nodes
87         foreach r in detaches do
88             delete(r→anti)
89             r→anti=NULL
90         detaches.clear()
91     foreach p in allocs do
92         p→setup(PAYLOAD,NULL,e_curr)
93     e_last=e_curr
94     e_curr=0
95 Function reset_op()
96     abort_op(true)
97     begin_op(true)
98 Function advance()
99     e=global_epoch.load()
100    foreach t in mindi whose val==e-1 do
101        TBP[t][(e-1)%4].pop_all()
102        clwb(descs[t])
103        update t's val to e in mindi
104    sfence
105    if some op linearized in e-1 or e then
106        global_epoch.CAS(e,e+1)
```

Figure 2 nbMontage pseudocode.

whose effects may not yet have been seen by other threads. In the wake of a crash in epoch e, nbMontage runs through the Ralloc heap, assembling a set of potentially allocated blocks and finding all CAS descriptors (identified by their `type` fields – lines 31 and 42). By matching the serial numbers and thread IDs of blocks and descriptors, the nbMontage-internal recovery procedure identifies all and only the payloads that are known, as of the crash, to have been in use at the end of epoch $e-2$. Specifically, if block B has thread ID t, serial number s, and epoch tag f, nbMontage will recover B if and only if

1. $0 < f \leq e-2$;
2. $(s < \texttt{descs}[t].\texttt{sn}) \vee (s = \texttt{descs}[t].\texttt{sn} \wedge \texttt{descs}[t].\texttt{status} = \texttt{COMMITTED})$; and
3. if B is a payload, it has not been canceled by an in-use anti-node.

Once the in-use blocks have been identified, nbMontage returns them to a data-structure-specific recovery routine that rebuilds any needed transient state, after which the state of the structure is guaranteed to reflect some valid linearization of pre-crash execution through the end of epoch $e - 2$.

3.5 Buffering Containers

Persistent blocks created or deleted in a given epoch will be recorded in thread- and epoch-specific to-be-persisted (TBP) and to-be-freed (TBF) containers. Every thread maintains four statically allocated instances of each (only 3 are actually needed, but indexing is faster with 4 – Fig. 2, lines 28 and 53).

TBPs are fixed-size circular buffers. When a buffer is full, its thread removes and writes back a block before inserting a new one. In the original version of Montage, epoch advance always occurs in a dedicated background thread (the `sync` operation handshakes with this thread). As part of the advance from epoch e to $e + 1$, the background thread iterates over all worker threads t, waits for t to finish any active operation in $e - 1$, extracts all blocks from TBP$[t][(e - 1) \bmod 4]$, and writes those blocks back to memory.

Insertions and removals from a TBP buffer never occur concurrently in the original version of Montage. In nbMontage, however, an operation that is lagging behind may not yet realize that it is doomed to retry, and may still be inserting items into the buffer when another thread (or several other threads!) decide to advance the epoch. The lagging thread, moreover, may even be active in epoch $e - 1 - 4k$, for some $k > 0$ (lines 88 and 92). This concurrency implies that TBPs need to support single-producer-multiple-consumers (SPMC) concurrency. Our implementation of the SPMC buffer (lines 1–21) maintains two monotonic counters, `pushed` and `popped`. To insert an item, a thread uses `store` to increment `pushed`. To remove some item(s), a thread uses `CAS` to increase `popped`. For simplicity, the code exploits the fact that duplicate writes-back are semantically harmless: concurrent removing threads may iterate over overlapping ranges (lines 13–21).

TBFs are dynamic-size, thread-unsafe containers implemented as vectors (line 53). Although deletion must respect epoch ordering, it can be performed lazily during crash-free execution, with each thread responsible for the blocks in its own TBFs. In `begin_op`, *after it has updated its descriptor*, thread t deletes blocks in TBF$[t][i \bmod 4]$, for $i \in [e_{last} - 1, \min(e_{last} + 1, e_{curr} - 2)]$, where e_{last} is the epoch of t's last operation and e_{curr} is the epoch of its current operation (lines 66–68).

3.6 Epoch Advance

To make `sync` nonblocking, we first decentralize the original epoch advance in Montage so that instead of making a request of some dedicated thread, every thread is now able to advance the epoch on its own. In the worst case, such an epoch advance may require iterating over the TBP buffers of all threads in the system. In typical cases, however, many of those buffers may be empty. To reduce overhead in the average case, we deploy a variant of Liu et al.'s *mindicator* [34] to track the oldest epoch in which any thread may still have an active operation. Implemented as a wait-free, fixed-size balanced tree, our variant represents each thread and its current epoch as a leaf. An ancestor in the tree indicates the minimum epoch of all nodes in its subtree. When thread t wishes to advance the epoch from e to $e + 1$, it first checks to see whether the root of the mindicator is less than e. If so, it scans up the tree from its own leaf until it finds an ancestor with epoch $< e$. It then reverses course, traces down the tree to find a lagging thread, persists its descriptor and any blocks in the requisite

TBP container, and repeats until the root is at least e. When multiple threads call `sync` concurrently, this nearest-common-ancestor strategy allows the work of persistence to be effectively parallelized. Experiments described in Section 5.3 confirm that our use of the mindicator, together with the lazy handling of TBF buffers (Section 3.5), leads to average `sync` times on the order of a few microseconds.

4 Correctness

We argue that nbMontage preserves the linearizability and lock freedom of a structure implemented on top of it, and adds buffered durable linearizability. We also argue that advances of the persistence frontier in nbMontage are wait free.

4.1 Linearizability

▶ **Theorem 1.** *Suppose that R is a data structure obeying the constraints of Section 3.2, running on nbMontage, and that K is realizable concrete history of R. K is linearizable.*

Proof (sketch). The `pnew`, `pdelete`, and `lin_CAS` routines of nbMontage have the same semantics as `new`, `delete`, and `CAS` calls in the original data structure. The `pdetach` routine has no semantic impact on crash-free execution: it simply ensures that a block whose removal has linearized will be reclaimed in post-crash recovery. The `sync` routine, similarly, has no semantic impact – with no parameters and no return values, it can linearize anywhere. If the instructions comprising each call to `pnew`, `pdelete`, `pdetach`, `sync`, and `lin_CAS` in a concrete nbMontage history are replaced with those of `new`, `delete`, `no-op`, `no-op`, and `CAS`, respectively, the result will be a realizable concrete history of the original data structure. Since that history is linearizable, so is the one on nbMontage. ◀

4.2 Buffered Durable Linearizability

As is conventional, we assume that each concurrent data structure implements some abstract data type. The semantics of such a type are defined in terms of legal *abstract sequential histories* – sequences of operations (request-response pairs), with their arguments and return values. We can define the *abstract state* of a data type, after some finite sequential history S, as the set of possible extensions to S permitted by the type's semantics. In a *concurrent* abstract history H, invocations and responses may be separated, and some responses may be missing, in which case the invocation is said to be *pending* at end of H. H is said to be *linearizable* if (1) there exists a history H' obtained by dropping some subset of the pending invocations in H and adding matching responses for the others, and (2) there exists a sequential history S that is equivalent to H' (same invocations and responses) and that respects both the *real-time* order of H' and the semantics of the abstract type. S is said to be a *linearization of* H.

Suppose now that R is a linearizable nonblocking implementation of type T, and that r is a concrete state of R – the bits in memory at the end of some concrete (instruction-by-instruction) history K. For R to be correct there must exist a mapping $\mathcal{M}$ such that for any such K and r, $\mathcal{M}(r)$ is the abstract state that results from performing, in order, the abstract operations corresponding to concrete operations that have linearized in K.

A structure R is *buffered durably linearizable* if post-crash recovery always results in some concrete state s that is justified by some prefix P of pre-crash concrete execution – that is, there exists a linearization S of the abstract history corresponding to P such that $\mathcal{M}(s)$ is the abstract state produced by S.

Consider again the 4 constraints listed at the end of Section 3.2 for data structures running on nbMontage. Elaborating on constraints 3 and 4, we use $r|_p$ to denote the set of payloads that were created (and inserted) and not yet detached by the operations that generated r. This allows us to recast constraint 4 and to add an additional constraint:

4′. There exists a mapping $\mathcal{Q}$ from sets of payloads to states of T such that for any concrete state r of R, $\mathcal{M}(r) = \mathcal{Q}(r|_p)$.

5. The recovery procedure of R, given a set of in-use payloads p, constructs a concrete state s such that $\mathcal{M}(s) = \mathcal{Q}(p)$.

▶ **Theorem 2.** *If a crash happens in epoch e, R will recover to a concrete state s such that $\mathcal{M}(s)$ is the abstract state produced by some linearization S of the abstract history H comprising pre-crash execution through the end of epoch $e-2$. In other words, R is buffered durably linearizable.*

Proof (sketch). For purposes of this proof, it is convenient to say that an update operation that commits the descriptor of its `lin_CAS` linearizes on the *preceding load* of the global epoch clock – the one that double-checks the clock before commit. Under this interpretation, if r is the concrete state of memory at the end of epoch $e-2$, we can say that $\mathcal{M}(r)$ reflects a sequential history containing all and only those operations that have committed their descriptors (line 17 in Fig. 8 in Full Version) by the end of the epoch. But this is not the only possible linearization of execution to that point! In particular, any operation that has loaded `global_epoch` (line 60 in Fig. 2), initialized its descriptor (line 65 of Fig. 2), and installed that descriptor in a `CASObj` (line 71 in Fig. 8 in Full Version) but has not yet committed the descriptor may "linearize in the past" (i.e., in epoch $e-2$) if it *or another, helping operation* commits the descriptor in the future. When a crash occurs in epoch e, any such retroactively linearized operations will see their payloads and anti-nodes included in the state s recovered from the crash. $\mathcal{M}(s)$ will therefore correspond, by constraint 5, to the linearization of execution through the end of epoch $e-2$ that includes all and only those pending operations that have linearized by the time of the crash. Crucially, if operation b depends on operation a, in the sense that a has completed in any extension of H in which b has completed, then, by constraint 2 of Section 3.2, the helping mechanism embodied by `lin_CAS` ensures that if b's payloads and anti-nodes are included in s, a's are included also. ◀

4.3 Wait-free Persistence

▶ **Theorem 3.** *The epoch advance in nbMontage is wait free.*

Proof (sketch). As shown in Fig. 2, an epoch advance from e to $e+1$ repeatedly finds a thread t that may still be active in $e-1$ (line 100), persists the contents of its TBP container and its descriptor (lines 101–102), and updates its mindicator entry. In the worst case, identifying all threads with blocks to be persisted requires time $O(T)$, where T is the number of threads, since the total size of the mindicator is roughly $2T$ nodes. Since each TBP container has bounded size, all the data of a thread can be persisted in $O(1)$ time. Mindicator updates, worst case, take $O(T \log T)$ time.

Since `sync` advances the epoch at most twice, it, too, is wait free. ◀

4.4 Lock freedom

▶ **Theorem 4.** *nbMontage preserves lock freedom during crash-free execution.*

Proof (sketch). Given Theorem 3, the only additional loop introduced by nbMontage is the automatic retry that occurs inside `lin_CAS` when the epoch has advanced. While this loop precludes wait freedom, we can (with a bit of effort – see App. C in Full Version) arrange to advance the epoch from e to $e + 1$ only if some update operation has linearized in epoch $e - 1$ or e (line 105 in Fig. 2). This suffices to preserve lock freedom. As a corollary, a data structure that is obstruction free remains so when persisted with nbMontage. ◀

5 Experimental Results

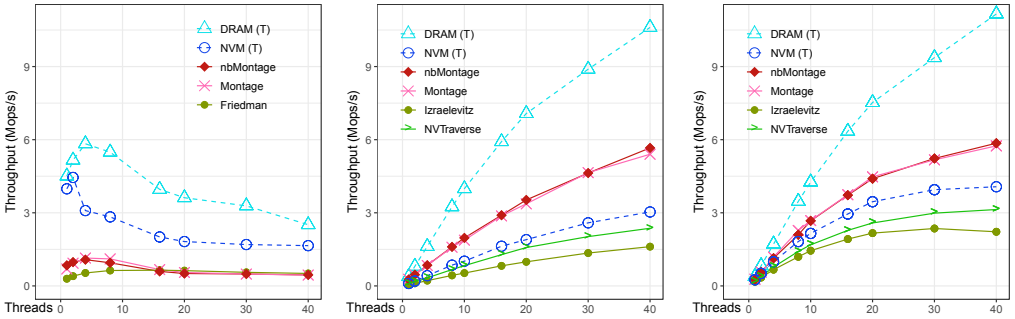
To confirm the performance benefits of buffering, we constructed nbMontage variants of Michael & Scott's queue [41], Natarajan & Mittal's binary search tree [42], the rotating skip list of Dick et al. [15], Michael's chained hash table [39], and the resizable hash table of Shalev & Shavit [48]. Mappings keep their key-value pairs in payloads and their index structures in transient memory. The queue uses payloads to hold values and their order. Specifically, between its two loads of the queue tail pointer, the enqueue operation calls `fetch_add` on a global counter to obtain a serial number for the to-be-inserted value. We benchmarked those data structures and various competitors on several different workloads. Below are the structures and systems we tested:

- **Montage** and **nbMontage** – as described in previous sections.
- **Friedman** – the persistent lock-free queue of Friedman et al. [19].
- **Izraelevitz** and **NVTraverse** – the N&M tree, the rotating skip list, and Michael's hash table persisted using Izraelevitz's transform [28] and the NVTraverse transform [18], respectively.
- **SOFT** and **NVMSOFT** – the lock-free hash table of Zuriel et al. [55], which persists only semantic data. SOFT keeps a full copy in DRAM, while NVMSOFT is modified to keep and access values only in NVM. Neither supports `update` on existing keys.
- **CLevel** – The persistent lock-free hash table of Chen et al. [8].
- **Dalí** – the lock-based buffered durably linearizable hash table of Nawab et al. [43].
- **DRAM (T)** and **NVM (T)** – as a baseline for comparison, these are unmodified transient versions of our various data structures, with data located in DRAM and NVM, respectively.

5.1 Configurations

We configured Montage and nbMontage with 64-entry TBP buffers and an epoch length of 10 ms. In practice, throughput is broadly insensitive to TBP size, and remains steady with epochs as short as 100 µs. All experiments were conducted on an Intel Xeon Gold 6230 processor with 20 physical cores and 40 hyperthreads, six 128 GB Optane Series 100 DIMMs, and six 32 GB DRAMs, running Fedora 30 Kernel 5.3.7 Linux Server. Threads are placed first on separate physical cores and then on hyperthreads. NVM is configured through the `dax-ext4` file system in "App Direct" mode.

All experiments use JEMalloc [16] for transient allocation and Ralloc [2] for persistent allocation, with the exception of CLevel, which requires the allocator from Intel's PMDK [49]. All chained hash tables have 1 million buckets. The warm-up phase for mappings inserts 0.5 M key-value pairs drawn from a key space of 1 M keys. Queues are initialized with 2000 items. Unless otherwise specified, keys and values are strings of 32 and 1024 bytes, respectively. We report the average of three trials, each of which runs for 30 seconds. Source code for nbMontage and the experiments is available at `https://github.com/urcs-sync/Montage`.

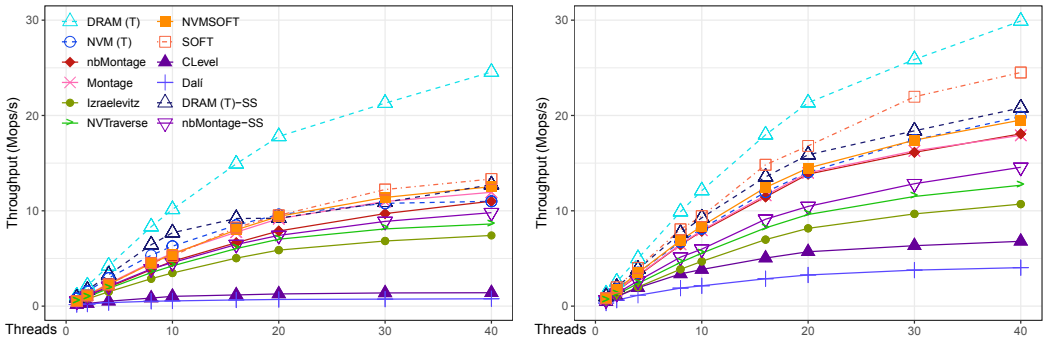


(a) Queues – 1:1 push:pop.

(b) Binary search trees – 2:1:1 get:insert:remove.

(c) Skip lists – 2:1:1 get:insert:remove.

Figure 3 Throughput of concurrent queues, binary search trees, and skip lists.



(a) 2:1:1 get:insert:remove.

(b) 18:1:1 get:insert:remove.

Figure 4 Hash table throughput. Options in the left column of the key are all variants of Michael's nonblocking algorithm.

5.2 Throughput

Results for queues, binary search trees, skip lists, and hash tables appear in Figures 3–4. The nbMontage versions of the M&S queue, N&M tree, rotating skip list, and Michael hash table outperform most persistent alternatives by a significant margin – up to 2× faster than Friedman et al.'s queue, 1.3–4× faster than NVTraverse and Izraelevitz et al.'s transform, and 3–14× faster than CLevel and Dalí. Significantly, nbMontage achieves almost the same throughput as Montage. SOFT and NVMSOFT are the only exceptions: the former benefits from keeping a copy of its data in DRAM; both benefit from clever internal optimizations. The DRAM copy has the drawback of forgoing the extra capacity of NVM; the optimization has the drawback of precluding atomic `update` of existing keys. While the transient Shalev & Shavit (S&S) hash table (DRAM (T)-SS in Fig. 4) is significantly slower than the transient version of Michael's hash table (DRAM (T)), the throughput of the Montage version (nbMontage-SS) is within 65% of the transient version and still faster than all other pre-existing persistent options other than SOFT and NVMSOFT.

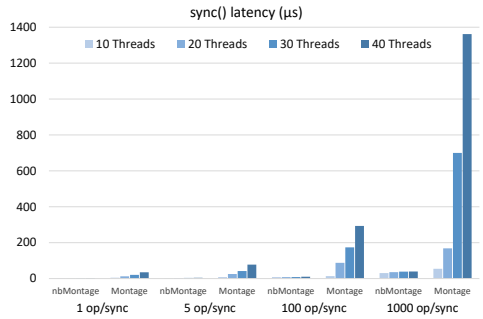


Figure 5 Average latency of `sync` on hash tables.

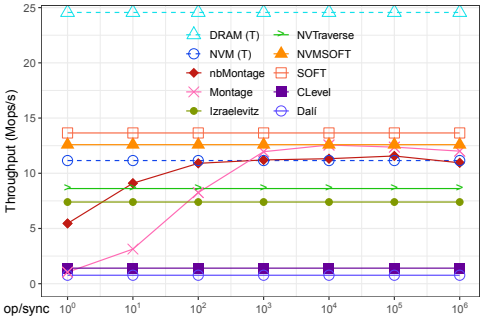


Figure 6 Throughput of hash tables with a `sync` every x operations on average.

5.3 Overhead of Sync

To assess the efficacy of nonblocking epoch advance and of our mindicator variant, we measured the latency of `sync` and the throughput of code that calls `sync` frequently. Specifically, using the nbMontage version of Michael's hash table, on the (2:1:1 get:insert:remove) workload, we disabled the periodic epoch advance performed by a background thread and had each worker call `sync` explicitly.

Average `sync` latency is shown in Figure 5 for various thread counts and frequencies of calls, on both nbMontage and its blocking predecessor. In all cases, nbMontage completes the typical `sync` in 1–40 µs. In the original Montage, however, `sync` latency never drops below 5 µs, and can be as high as 1.3 ms with high thread count and low frequency.

Hash table throughput as a function of `sync` frequency is shown in Figure 6 for 40 threads running on Montage and nbMontage. For comparison, horizontal lines are shown for various persistent alternatives (none of which calls `sync`). Interestingly, nbMontage is more than 2× faster than CLevel even when `sync` is called in every operation, and starts to outperform NVTraverse once there are more than about 10 operations per `sync`.

5.4 Recovery

To assess recovery time, we initialized the nbMontage version of Michael's hash table with 1–32 M 1 KB elements, leading to a total payload footprint of 1–32 GB. With one recovery thread, nbMontage recovered the 1 GB data set in 1.4 s and the 32 GB data set in 103.8 s (22.2 s to retrieve all in-use blocks; 81.6 s to insert them into a fresh hash table). Eight recovery threads accomplished the same tasks in 0.3 s and 17.9 s. These times are all within 0.5 s of recovery times on the original Montage.

6 Conclusions

To the best of our knowledge, nbMontage is the first general-purpose system to combine buffered durable linearizability with a simple API for nonblocking data structures and nonblocking progress of the persistence frontier. Nonblocking persistence allows nbMontage to provide a very fast wait-free `sync` routine and to strictly bound the work that may be lost on a crash. Lock-free and wait-free structures, when implemented on nbMontage, remain lock free; obstruction-free structures remain obstruction free.

Experience with a variety of nonblocking data structures confirms that they are easy to port to nbMontage, and perform extremely well – better in most cases than even hand-built structures that are strictly durably linearizable. 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. In ongoing work, we are exploring the design of a hybrid system that supports both lock-based and nonblocking structures, with nonblocking persistence in the absence of lock-based operations. We also hope to develop a buffered durably linearizable system for object-based software transactional memory, allowing persistent operations on multiple data structures to be combined into failure-atomic transactions.

References

1. H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott. Nonblocking persistent software transactional memory. In *27th Intl. Conf. on High Performance Computing, Data, and Analytics (HiPC)*, pages 283–293, 2020. doi:10.1109/HiPC50609.2020.00042.
2. Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. Understanding and optimizing persistent memory allocation. In *19th Intl. Symp. on Memory Management (ISMM)*, 2020. doi:10.1145/3381898.3397212.
3. Hokeun Cha, Moohyeon Nam, Kibeom Jin, Jiwon Seo, and Beomseok Nam. B3-tree: Byte-addressable binary B-tree for persistent memory. *ACM Trans. on Storage*, 16(3):17:1–17:27, July 2020. doi:10.1145/3394025.
4. Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 433–452, Portland, OR, October 2014. doi:10.1145/2660193.2660224.
5. Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. of the VLDB Endowment*, 8(5):497–508, 2015. doi:10.14778/2735479.2735483.
6. Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. of the VLDB Endowment*, 8(7):786–797, February 2015. doi:10.14778/2752939.2752947.
7. Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: A persistent B+-tree with low tail latency. *Proc. of the VLDB Endowment*, 13(11):2634–2648, August 2020. doi:10.14778/3407790.3407850.
8. Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *Usenix Annual Technical Conf. (ATC)*, pages 799–812, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/chen.
9. Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven 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. doi:10.1145/1950365.1950380.
10. Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. In *24th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 441–454, Providence, RI, USA, 2019. doi:10.1145/3297858.3304046.
11. Nachshon Cohen, David T. Aksun, and James R. Larus. Object-oriented recovery for non-volatile memory. *Proc. of the ACM on Programming Languages*, 2(OOPSLA):153:1–153:22, October 2018. doi:10.1145/3276523.
12. Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *30th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, page 259–269, Vienna, Austria, 2018. doi:10.1145/3210377.3210400.
13. Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *30th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 271–282, Vienna, Austria, July 2018. doi:10.1145/3210377.3210392.

14 Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Usenix Annual Technical Conf. (ATC)*, pages 373–386, Boston, MA, 2018. URL: https://www.usenix.org/conference/atc18/presentation/david.

15 Ian Dick, Alan Fekete, and Vincent Gramoli. A skip list for multicore. *Concurrency and Computation: Practice and Experience*, 29(4), May 2016.

16 Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conf.*, Ottawa, ON, Canada, May 2006. URL: https://papers.freebsd.org/2006/bsdcan/evans-jemalloc.files/evans-jemalloc-paper.pdf.

17 Keir Fraser. *Practical Lock-Freedom.* PhD thesis, King's College, Univ. of Cambridge, 2003. Published as Univ. of Cambridge Computer Laboratory technical report #579, February 2004. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

18 Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *41st ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 377–392, 2020. doi:10.1145/3385412.3386031.

19 Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, Vienna, Austria, 2018. doi:10.1145/3178487.3178490.

20 Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, HTM-compatible persistent transactions. In *41st ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 59–74, June 2020. doi:10.1145/3385412.3385991.

21 Ellis R. Giles, Kshitij Doshi, and Peter Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *31st Symp. on Mass Storage Systems and Technologies (MSST)*, pages 1–14, Santa Clara, CA, May–June 2015. doi:10.1109/MSST.2015.7208276.

22 Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Usenix Annual Technical Conf. (ATC)*, pages 913–928, Renton, WA, July 2019. URL: https://www.usenix.org/conference/atc19/presentation/gu.

23 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *16th Intl. Symp. on Distributed Computing (DISC)*, pages 265–279, Toulouse, France, October 2002. doi:10.1007/3-540-36108-1_18.

24 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

25 Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *12th European Conf. on Computer Systems (EuroSys)*, pages 468–482, Belgrade, Serbia, 2017. doi:10.1145/3064176.3064204.

26 Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *16th Usenix Conf. on File and Storage Technologies (FAST)*, pages 187–200, Oakland, CA, February 2018. URL: https://www.usenix.org/conference/fast18/presentation/hwang.

27 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 427–442, Atlanta, GA, 2016. doi:10.1145/2872362.2872410.

28 Joseph Izraelevitz, Hammurabi Mendes, and Michael 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, September 2016. doi:10.1007/978-3-662-53426-7_23.

29 Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. clfB-tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Trans. on Storage*, 14(1):5:1–5:17, February 2018. doi:10.1145/3129263.

30 Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th Usenix Conf. on File and Storage Technologies (FAST)*, pages 257–270, Santa clara, CA, February 2017. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon.

31 Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. Building scalable NVM-based B+tree with HTM. In *48th Intl. Conf. on Parallel Processing (ICPP)*, pages 101:1–101:10, Kyoto, Japan, August 2019. doi:10.1145/3337821.3337827.

32 Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–343, Xi'an, China, April 2017. doi:10.1145/3037697.3037714.

33 Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Intl. Symp. on Microarchitecture (MICRO)*, pages 258–270, Fukuoka, Japan, 2018. doi:10.1109/MICRO.2018.00029.

34 Yujie Liu, Victor Luchangco, and Michael Spear. Mindicators: A scalable approach to quiescence. In *IEEE 33rd Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 206–215, Philadelphia, PA, 2013. doi:10.1109/ICDCS.2013.39.

35 Pratyush Mahapatra, Mark D Hill, and Michael M Swift. Don't persist all: Efficient persistent data structures, 2019. arXiv preprint. arXiv:1905.13011.

36 Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *7th European Conf. on Computer Systems (EuroSys)*, pages 183–196, Bern, Switzerland, April 2012. doi:10.1145/2168836.2168855.

37 Amirsaman Memaripour, Joseph Izraelevitz, and Steven 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, March 2020. doi:10.1145/3373376.3378456.

38 Amirsaman Memaripour and Steven Swanson. Breeze: User-level access to non-volatile main memories for legacy software. In *36th Intl. Conf. on Computer Design (ICCD)*, pages 413–422, Hartford, CT, October 2018. doi:10.1109/ICCD.2018.00069.

39 Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *14th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 73–82, Winnipeg, MB, Canada, 2002. doi:10.1145/564870.564881.

40 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(6):491–504, June 2004. doi:10.1109/TPDS.2004.8.

41 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 267–275, Philadelphia, PA, 1996. doi:10.1145/248052.248106.

42 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, Orlando, FL, USA, February 2014. doi:10.1145/2555243.2555256.

43 Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A periodically persistent hash map. In *Intl. Symp. on Distributed Computing (DISC)*, pages 37:1–37:16, Vienna, Austria, 2017. doi:10.4230/LIPIcs.DISC.2017.37.

44 Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Intl Conf on Management of Data (SIGMOD)*, pages 371–386, San Francisco, CA, 2016. doi:10.1145/2882903.2915251.

45 Matej Pavlovic, Alex Kogan, Virendra J Marathe, and Tim Harris. Brief announcement: Persistent multi-word compare-and-swap. In *ACM Symp. on Principles of Distributed Computing (PODC)*, pages 37–39, Egham, United Kingdom, 2018. doi:10.1145/3212734.3212783.

46 Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. OneFile: A wait-free persistent transactional memory. In *49th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 151–163, Portland, OR, June 2019. doi:10.1109/DSN.2019.00028.

47 David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *3rd VLDB Workshop on In-Memory Data Management and Analytics (IMDM)*, pages 4:1–4:8, Kohala, HI, 2015. doi:10.1145/2803140.2803144.

48 Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, pages 379–405, May 2006. doi:10.1145/1147954.1147958.

49 Usharani Upadhyayula and Andy M. Rudoff. Introduction to Programming with Persistent Memory from Intel. https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel, August 2017.

50 Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *9th Usenix Conf. on File and Storage Technologies (FAST)*, pages 61–75, San Jose, CA, 2011. URL: http://www.usenix.org/events/fast11/tech/techAbstracts.html#Venkataraman.

51 Haris Volos, Andres Jaan Tack, and Michael 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, 2011. doi:10.1145/1950365.1950379.

52 Chundong Wang, Qingsong Wei, Lingkun Wu, Sibo Wang, Cheng Chen, Xiaokui Xiao, Jun Yang, Mingdi Xue, and Yechao Yang. Persisting RB-Tree into NVM in a consistency perspective. *ACM Trans. on Storage*, 14(1):6:1–6:27, February 2018. doi:10.1145/3177915.

53 Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. A fast, general system for buffered persistent data structures. In *50th Intl. Conf. on Parallel Processing (ICPP)*, August 2021. URL: https://oaciss.uoregon.edu/icpp21/views/includes/files/pap118s4-file2.pdf.

54 Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *13th Usenix Conf. on File and Storage Technologies (FAST)*, pages 167–181, Santa Clara, CA, February 2015. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang.

55 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proc. of the ACM on Programming Languages*, 3(OOPSLA):128:1–128:26, October 2019. doi:10.1145/3360554.

A Example nbMontage Data Structure

As an example of using the nbMontage API, Fig. 7 presents a fragment of Michael's lock-free hash table [39], modified for persistence. Highlighted parts were changed from the original.

```
class MHashTable :  public Recoverable {
class Payload :  public PBlk { K key; V val; };
struct Node { // Transient index class
  Payload* payload = nullptr; // Transient-to-persistent pointer
  CASObj<Node*> next = nullptr; // Transient-to-transient pointer
  Node(K& key, V& val) { payload = pnew<Payload>(key, val); }
  ~Node() { if(payload!=nullptr) pdelete(payload); }
};
EBRTracker tracker; // Epoch-based memory reclamation
bool find(CASObj<Node*>* &p,Node* &c,Node* &n,K k); // Starting from p, find node >= k and assign to c
void put(K key, V val) { // Insert, or update if the key exists
  Node* new_node = new Node(key, val);
  CASObj<Node*>* prev = nullptr;
  Node* curr;
  Node* next;
  tracker.start_op();
  while(true) {
    if (find(prev,curr,next,key)) { // update
      new_node->next.store(curr);
      pdetach(curr->payload);
      if(prev->lin_CAS(curr,new_node)) {
        while(!curr->next.CAS(next,mark(next))) next=curr->next.load();
        if(new_node->next.CAS(curr,next)) tracker.retire(curr);
        else find(prev,curr,next,key);
        break;
      }
    } else { // key does not exist; insert
      new_node->next.store(curr);
      if(prev->lin_CAS(curr,new_node))
        break;
    }
  }
  tracker.end_op();
};
```

Figure 7 Michael's lock-free hash table example (nbMontage-related parts highlighted).