A Fast, General System for Buffered Persistent Data Structures

Haosen Wen[*]
Wentao Cai[*]
{hwen5,wcai6}@cs.rochester.edu

Mingzhe Du
Louis Jenkins
{mdu5,ljenkin4}@cs.rochester.edu
University of Rochester
Rochester, New York, USA

Benjamin Valpey
Michael L. Scott
{bvalpey,scott}@cs.rochester.edu

ABSTRACT

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

We present what we believe to be the first general-purpose approach to building *buffered* persistent data structures, and a system, Montage, to support that approach. Montage is built on top of the Ralloc nonblocking persistent allocator. It employs a millisecond-granularity *epoch clock*, and ensures that no operation appears to span an epoch boundary. It also arranges to persist only that data minimally required to reconstruct the structure after a crash. If a crash occurs in epoch e, all work performed in epochs e and $e-1$ is lost, but work from prior epochs is preserved, consistently. As in traditional file and database systems, a `sync` operation can be used to flush buffers on demand; the Montage `sync` is extremely fast.

We describe the implementation of Montage, argue its correctness, and report unprecedented throughput for persistent queues, sets/mappings, and general graphs.

CCS CONCEPTS

• **Theory of computation** → **Parallel computing models**; • **Computing methodologies** → **Concurrent algorithms**; • **Computer systems organization** → **Reliability**.

KEYWORDS

Buffered Durable Linearizability, Data Structures, Consistency

ACM Reference Format:
Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. 2021. A Fast, General System for Buffered Persistent Data Structures . In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472456.3472458

[*]The first two authors contributed equally to this work.

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472458

1 INTRODUCTION

Emerging memory technologies such as Intel's *Optane* are significantly denser and less power hungry than traditional DRAM. While such memory could simply be used as a plug-in replacement for DRAM, its *nonvolatility* also raises the intriguing possibility of keeping long-lived data in pointer-rich "in memory" format across program runs and even system crashes, rather than serializing to and from a file system or back-end database.

Crashes cause problems, however. For file systems and databases, long-established logging techniques ensure that transitions from one consistent state to another are failure atomic. For data structures accessed with load and store instructions, the cost of such logging may be prohibitively high. Moreover, the fact that caches remain volatile (at least on current processors) and may write back their contents out of program order means that data structure operations must typically issue explicit write-back and fence instructions to guarantee post-crash consistency.

Past work has established *durable linearizability* as the standard correctness criterion for persistent data structures [14, 21, 32, 50]. This criterion builds on the familiar notion of linearizability for concurrent (non-persistent) structures. A structure is said to be linearizable if operations that may overlap in time always have the same effect as some one-at-a-time execution that respects both "real time" order (if operation A returns before operation B is called, then A must appear to happen before B) and the semantics of the abstraction represented by the structure.

A persistent data structure is said to be *durably linearizable* if (1) it is linearizable during crash-free operation, (2) each operation persists (reaches a state that will survive a crash) between its call and return, and (3) the order of persists matches the linearization order. These semantics, however, are significantly stronger than most programs need or most programmers expect. A file or database operation, after all, returns to its caller while data remain in volatile DRAM buffers. An operation that requires synchronous persistence—e.g., before responding to a client over the network—performs a `sync` operation. A data structure that mimics this more conventional, relaxed persistence is said to be *buffered* durably linearizable. Like a file or database system, it guarantees on a crash to preserve some consistent prefix of pre-crash execution.

Recent publications have described many individual durably linearizable data structures and perhaps two dozen general-purpose systems to provide failure atomicity for outermost critical sections or speculative transactions (Sec. 2). To the best of our knowledge, all of the general-purpose systems and all but two of the individual structures (the Dalí hashmap [35] and InCLL MassTree [8]) are strictly durably linearizable. To reduce the overhead of synchronous persistence and to provide more conventional semantics, we present what we believe to be the first general-purpose approach to buffered durably linearizable structures.[1] Our system, *Montage*, employs a global *epoch clock*, and ensures that no operation appears to span an epoch boundary. If a crash occurs in epoch e, Montage recovers the state of the abstraction from the end of epoch $e-2$.

Generalizing an approach embodied in several previous data structures [36, 49, 50], Montage also distinguishes between the *abstract* (semantic) state of the concurrent object and its *concrete* (implementation-level) state. It encourages the programmer to maintain only the former in NVM, to reduce persistence overhead. A Montage mapping, for example, would typically persist only a bag of key-value pairs; the look-up structure (hash table, tree, skip list) would live entirely in transient DRAM. During recovery, Montage cooperates with the user program to rebuild the concrete state.

Our implementation of Montage is built on top of Ralloc [3], a lock-free allocator for persistent memory. Montage itself is also lock-free during normal operation, though a stalled thread can arbitrarily delay progression of the persistence frontier. Performance experiments (Sec. 6) reveal that a Montage hashmap on a 2-socket server can sustain well over 20 M ops/s on a read-heavy workload—7× as many as the Dalí hashmap, 17× as many as the state-of-the-art Pronto system [32], and within a factor of 3 of a transient DRAM table. This is close to the best one could hope for: read bandwidth for Intel Optane NVM is about one-third that of DRAM [22].

Summarizing contributions, we present: (1) The first general system, Montage, for buffered durably linearizable structures; (2) informal proofs of safety and liveness; and (3) performance results for a variety of data structure microbenchmarks, the `memcached` key-value store, and a general library for graphs. Relative to the state of the art in both general systems and special-purpose structures, we obtain unprecedented throughput without significantly compromising recovery times.

2 RELATED WORK

Recent years have seen an explosion of work on persistent data structures, much of it focused on B-tree indices for file systems and databases [6, 19, 34, 36, 43, 49]. Other work has targeted RB trees [45], radix trees [26], hashmaps [35, 41, 50], and queues [14]. Several projects persist only parts of a data structure, and rebuild the rest on recovery. Zuriel et al. [50] argue that this approach can be used for almost any implementation of a set or mapping. Unfortunately, their SOFT system keeps a full copy of the data in DRAM, forfeiting the high capacity of NVM, and fails to support atomic `update`. Montage eliminates these restrictions; it also supports any abstraction that comprises items and relationships—effectively, anything that can be represented as a graph.

Several existing data structures are designed to linearize by using a single compare-and-swap (CAS) instruction to replace a portion of the structure [6, 26, 34, 35]. If the new portion is persisted before the CAS, and the updated pointer is persisted immediately after the CAS, no separate logging is required. Mahapatra et al. [31] and Haria et al. [17] apply this observation to a variety of "functional" data structures, building sets, maps, stacks, queues, and vectors. As an extension, a sequence of single-CAS steps can be used to move a structure through self-documenting intermediate stages [19, 45].

Izraelevitz et al. [21] provide a mechanical construction to convert any nonblocking concurrent structure into a correct persistent version. David et al. [12] describe several techniques to eliminate redundant writes-back and fences for such structures, significantly improving performance.

Many groups now have developed systems to ensure the failure atomicity of lock-based critical sections [4, 18, 20, 29, 46, 47] or speculative transactions [2, 5, 7, 9, 11, 15, 16, 24, 33, 37, 38, 42, 44]. Significantly, *all* of these systems ensure that an operation has persisted before permitting the calling thread to proceed—that is, they adopt the strict version of durable linearizability.

The Dalí hashmap [35] delays persistence, so the overhead of writes-back and fencing can be amortized over many operations while still providing *buffered* durable linearizability. The implementation relies on a flush-the-whole-cache instruction that is available only in privileged mode on the x86, and has the side effect of evicting many useful lines. Similarly, Cohen et al. [8] embed undo logs inside every cache line and periodically flush the entire cache; their technique is inapplicable to large values spanning a cache line. Our reimplementation of Dalí (used in Sec. 6) tracks to-be-written-back lines explicitly in software—as does Montage. Montage then extends delayed persistence to arbitrary structures.

Perhaps the closest prior work to Montage is the Pronto system of Memaripour et al. [32], which logs *high level* operations (rather than low-level updates), and replays the log after a crash. Periodic checkpoints allow it to bound the length of the log, and thus recovery time. Notably, Pronto still pays the cost of persisting each operation before returning; extending Pronto to buffer its updates would be a highly nontrivial change. TimeStone [24], likewise, combines high-level logging and periodic checkpointing, but the fact that it keeps multiple versions of each object in DRAM means that, like SOFT, it is unable to make full use of NVM capacity.

3 MONTAGE DESIGN

Montage manages persistent *payload* blocks on behalf of one or more concurrent data structures. A programmer who wishes to adapt a structure to Montage must identify the subset of the data that is needed, in quiescence, to capture the state of the abstraction. A set, for example, needs to keep its items in payload blocks, but not its lookup structure. A mapping needs to keep key-value pairs. A queue needs to keep its items *and* their order: it might label payloads with consecutive integers from i (the head) to j (the tail). A graph can keep a payload for each vertex (each with a unique name) and a payload for each edge (each of which names two vertices).

A typical data structure maintains additional, transient indexing data to speed up retrievals. A set or mapping might maintain a hash table, tree, or skip list. A queue might maintain a linked list of pointers to items. A graph might maintain a transient object for each vertex, containing a pointer to a payload for the vertex attributes, a set of pointers to neighboring vertex objects, and (if edges have large attributes) a set of pointers to edge payloads. All of this transient data can be reconstructed after a crash.

Crucially, synchronization may safely be performed on transient data. *Montage does not, itself, determine the linearization order of*

[1] We have recently become aware of the concurrently developed CpNvm system [1], which is also buffered durably linearizable. Unlike Montage, CpNvm duplicates the entire data structure in DRAM and NVM, updating the NVM copy at epoch boundaries.

```cpp
namespace pds{
  class PBlk;           // Base class for payloads
  // Macro to generate get() and set() methods for field
  // fieldname of type type_name within payload_type
  GENERATE_FIELD(type_name, fieldname, payload_type);
  // Creates `protected m_fieldname` with the following members:
    // get value with old-see-new alert enabled
    const type_name& get_fieldname();
    // get with old-see-new alert disabled
    const type_name& get_unsafe_fieldname();
    // set value of fieldname; may return a new payload
    payload_type* set_fieldname(type_name&);

  class EpochSys;
  class Recoverable{    // Base class for Montage structures
    // Instance of this structure's epoch system
    EpochSys* esys;
    // Begin op in current epoch; mark already-created payloads
    void BEGIN_OP();
    // End an operation
    void END_OP();
    // Begin a scoped operation using RAII
    BEGIN_OP_AUTOEND();
    // Create a payload block
    payload_type* PNEW(payload_type, ...);
    // Delete a payload after end of next epoch
    void PDELETE(PBlk*);
    // Throw exception if epoch has changed
    CHECK_EPOCH();
    // Request and wait for two-epoch advance
    void sync();
  };
  struct OldSeeNewException : public std::exception;
};
```

Figure 1: C++ API.

operations. Rather it ensures that the persistence order for payloads is consistent with the linearization order of the underlying structure. More specifically, it divides execution into *epochs* in such a way that every epoch boundary represents a consistent cut of the happens-before relationship among operations; it then arranges, in the wake of a crash, to recover all managed data structures to their state as of some common epoch boundary.

3.1 API

The Montage C++ API is shown in Figure 1. A lock-based hashmap built with Montage appears in Figure 2.

Any data structure operation that creates or updates payloads must make itself visible to Montage by calling `BEGIN_OP`. It indicates completion with `END_OP`. For ease of use, Montage also provides `BEGIN_OP_AUTOEND`, which uses the RAII idiom to call `BEGIN_OP` immediately and to call `END_OP` automatically at the end of the current scope. Read-only operations can skip these calls, though they must still synchronize on the transient data structure. Payloads are created and destroyed using `PNEW` and `PDELETE`. Existing payloads are accessed with `get` and `set` methods, created by the `GENERATE_FIELD` macro; `get` returns a `const` reference to the field; `set` updates the field and returns a (possibly altered) pointer to the payload.

To support the epoch system, Montage labels all payloads with the epoch in which they were created *or most recently modified.* An operation in epoch e that wishes to modify an existing payload can do so "in place" if the payload was created in e; otherwise, Montage creates a *new* payload with which to replace it. The `set` methods

```cpp
class HashMap : public Recoverable{
  // Payload class
  class Payload : public PBlk{
    GENERATE_FIELD(K, key, Payload);
    GENERATE_FIELD(V, val, Payload);
  };
  struct ListNode{    // Transient index class
    // Transient-to-persistent pointer
    Payload* payload = nullptr;
    // Transient-to-transient pointers
    ListNode* next = nullptr;
    void set_val_wrapper(V& v){
      payload = payload->set_val(v);
    }
    ListNode(K& key, V& val){
      payload = PNEW(Payload, key, val);
    }
    ~ListNode(){
      PDELETE(payload);
    }
    // get() methods omitted
  };
  // Insert, or update if the key exists
  optional<V> put(K key, V val, int tid){
    size_t idx=hash_fn(key)%idxSize;
    ListNode* new_node = new ListNode(key, val);
    std::lock_guard lk(buckets[idx].lock);
    BEGIN_OP_AUTOEND();
    ListNode* curr = buckets[idx].head.next;
    ListNode* prev = &buckets[idx].head;
    while(curr){
      K& curr_key = curr->get_key();
      if (curr_key == key){
        optional<V&> ret = curr->get_val();
        curr->set_val_wrapper(val);
        delete new_node;
        return ret;
      } else if (curr_key > key){
        new_node->next = curr;
        prev->next = new_node;
        return {};
      } else {
        prev = curr;
        curr = curr->next;
      }
    } // while
    prev->next = new_node;
    return {};
  }
};
```

Figure 2: Simple lock-based hashmap (Montage-related parts highlighted).

enforce this convention by returning a pointer to a new or copied payload, as appropriate.

During a given epoch, "hot" payloads will typically be modified in place. When a new copy is created, however, an operation must rewrite any pointers to the payload found anywhere in the structure. For this reason, it is important to minimize the number of pointers to a given payload found in transient data; this can be trivially accomplished by indirecting all such pointers through a transient intermediate object. It is even more important to avoid long chains of pointers in persistent data: otherwise, a change to payload p, at the end of a long chain, would require a change to the penultimate payload p', which would in turn require a change to its predecessor p'', and so on.

Because calls to `get` are invisible to recovery, they can safely be made outside the bounds of `BEGIN_OP` and `END_OP` (subject to transient synchronization). Calls to PNEW can also be made early; the payloads they return will automatically be recorded and properly labeled when BEGIN_OP is called.

3.2 Periodic Persistence

The key task of Montage is to ensure that operations persist atomically, in an order consistent with their linearization order. Toward that end, the system ensures that

(1) all payloads created or modified by a given operation are labeled with the same epoch number;
(2) all payloads created or modified in a given epoch e persist together, instantaneously, when the epoch clock ticks over from $e + 1$ to $e + 2$; and
(3) each update operation linearizes in the epoch in which it created payloads.

Property 1 is ensured by the set and PNEW methods, as described in Section 3.1. Note that an operation that begins in epoch e can continue to create and modify payloads in that epoch, even if the clock ticks over to $e + 1$.

Property 2 is enforced by Montage's recovery routines: if a crash occurs in epoch e, those routines discard all payloads labeled e or $e - 1$, but keep everything that is older. This two-epoch convention, as suggested by Nawab et al. [35], allows operations in e and $e - 1$ to overlap in time, avoiding the need for quiescence on clock ticks. At the same time, it requires that memory reclamation be delayed. If a payload created or updated in epoch b is passed to PDELETE in epoch $e > b$, Montage creates an "anti-payload" labeled e. If a crash occurs before $e + 2$, the anti-payload will be discarded and the original payload retained. If a crash occurs during epoch $e + 2$, the anti-payload will be discovered during recovery and both it and the original payload will be discarded. If execution proceeds without a crash, the original payload will be reclaimed when the epoch advances from $e + 2$ to $e + 3$; the anti-payload will be reclaimed when the epoch advances from $e + 3$ to $e + 4$.

Property 3 is the responsibility of the transient data structure built on top of Montage. Lock-based operations are easy: no conflicting operation can proceed until we release our locks, and we can easily pretend that all updates happened at the last call to set or PNEW. For nonblocking structures, a similar guarantee can be made if every operation linearizes on a statically identified compare-and-swap (CAS) instruction that also modifies an adjacent counter (as is often used to avoid ABA anomalies). One first reads some variable x, verifies the epoch clock (using the CHECK_EPOCH method), and only then attempts a CAS on x. If the CAS succeeds, it can be said to have occurred at the time of the CHECK_EPOCH call. This strategy generally requires read-only operations on the structure to be modified by replacing their linearizing read with a read-CAS primitive (wrapped as load_verify1 in Montage) that updates the adjacent count: otherwise a read that occurs immediately after an epoch change might observe an update from the previous epoch as not yet having occurred. For cases in which this modification is undesirable (e.g., because reads vastly outnumber updates), we use a variant of the double-compare-single-swap (DCSS) software primitive of Harris et al. [25] (wrapped as CAS_verify2) to update a location while simultaneously verifying the current epoch number. A compatible read primitive (load_verify2) performs no store instructions (and thus induces no cache evictions) so long as no DCSS is currently in progress on the variable being read; if one is, the read helps the DCSS complete.

As an assist to programmers in ensuring property 3, Montage raises an exception called OldSeeNewException whenever an operation running in epoch e reads a payload created in some epoch $e' > e$. In most cases, programmers can ensure that this exception will never arise. In other cases, the operation may respond to the exception by rolling back what has done so far and starting over in the newer epoch. In special cases, an operation can ignore the exception or use get_unsafe methods to avoid generating it in the first place (the new data might, for example, be used only for semantically neutral performance enhancement).

In support of these properties, the epoch-advancing mechanism at the end of epoch e (1) waits until no operation is active in epoch $e - 1$; (2) reclaims all payloads deleted in epoch $e - 2$ and all anti-payloads created in epoch $e-3$; (3) explicitly writes back all payloads created or modified in epoch $e - 1$; (4) waits for the writes-back to complete; and (5) updates and writes back the epoch clock. Further details appear in Section 5.

3.3 Nonblocking Data Structures

As described in Section 3.2, Montage is compatible with nonblocking operations that employ special CAS or load primitives to ensure that linearization occurs in the epoch in which any payloads were created or modified. In the general case, a structure that uses the OldSeeNewException to keep its linearization order consistent with epoch order may find that the resulting restarts make it lock-free or obstruction-free, rather than wait-free. Still, nothing in Montage precludes lock freedom.

4 CORRECTNESS

We argue that Montage (1) preserves, during crash-free operation, the linearizability of a structure implemented on top of it, (2) adds buffered durable linearizability, and (3) preserves lock freedom.

Each concurrent data structure serves to implement some abstract data type. The semantics of such a type are defined in terms of *legal histories*—sequences of operations, with their arguments and return values. The implementation is correct if it is *linearizable*, meaning that every concurrent history (with overlapping calls and returns from different threads) is equivalent to (has the same operations as) some sequential history that is consistent with real-time order (if A returns before B is called in the concurrent history, then A precedes B in the sequential history) and that represents a valid operation sequence for the data type.

We can define the abstract *state* of a data type, after a finite sequence of operations, as the set of sequences that are permitted to extend that sequence according to the type's semantics. Suppose, then, that data structure S is a correct implementation of data type T, and that s is a quiescent concrete state of S (the bits in memory at some point when no operations are active). We can define the *meaning* of that state, $\mathcal{M}(s)$, as the state of T after the sequence of abstract operations corresponding to (a linearization of) the operations performed so far on S.

We assume that the programmer using Montage obeys the following *well-formedness* constraints:

(1) Each data structure S, implemented on top of Montage, is linearizable when Montage itself is disabled and crashes do not occur. More specifically, assume that (a) `PNEW` and `PDELETE` are implemented as ordinary `new` and `delete`; (b) `get` and `set` are ordinary accessor methods, and `set` never copies a payload; (c) `BEGIN_OP` and `END_OP` are no-ops; and (d) the `OldSeeNewException` never arises. Under these circumstances, the structure is linearizable.
(2) Any synchronization required for linearizability is performed solely on transient data: accesses to payloads, which may be replaced on an update, never participate in a data *or* synchronization race.
(3) All accesses to payloads are made through `get` and `set`. Each operation that modifies the data structure (a) calls `BEGIN_OP` before `set`, (b) calls `END_OP` after completing all its `sets`, and (c) ensures that between its last call to `set` or `CHECK_EPOCH` and its linearization point, no conflicting operation can linearize.
(4) Whenever `set` returns a pointer to a payload different than the one on which it was called, the calling operation replaces every pointer to the old payload in the structure with a pointer to the new payload. As noted in Section 3.1, this can be trivially accomplished by indirecting all such pointers through a transient intermediate object.
(5) There exists a mapping Q from sets of payloads to states of T such that whenever S is quiescent, $\mathcal{M}(s) = Q(p)$, where s is the concrete state of S and p is the current set of payloads.
(6) The recovery routine for S, given a set of payloads r, constructs a concrete state t such that $\mathcal{M}(t) = Q(r)$.

4.1 Linearizability

Lemma 4.1. *A well-formed, linearizable concurrent data structure, implemented on top of Montage, remains well-formed and linearizable when Montage is enabled.*

Proof (sketch). Constraint 4 ensures that any payload cloned by Montage is reattached to the structure wherever the old payload appeared. Since access to payloads is race-free (Constraint 2), this re-attachment is safe. Throws of the `OldSeeNewException` will be harmless: they simply facilitate compliance with Constraint 3; any operation that already satisfies that constraint can safely ignore the exception. Finally, given the mapping Q from payloads to abstract state (Constraint 5), we can easily create a Q' that ignores both the old versions of cloned payloads and any payloads for which an anti-payload exists. These are the only effects of enabling Montage that are visible to the structure during crash-free execution. □

Theorem 4.2. *A Montage data structure S remains linearizable when epoch advancing operations are added to its history.*

Proof (sketch). Let a_e denote the operation that advances the epoch from $e-1$ to e. Consider a linearization order for S itself, as provided by Lemma 4.1. Constraint 3 ensures that the linearization point of any update operation in this order occurs between events a_e and a_{e+1}, making it easy to place these events into the linearization order. A read-only operation, moreover, has no forward or anti-dependences on the epoch clock, so it cannot participate in any circular dependence with respect to the epoch advancing events. □

4.2 Buffered Durable Linearizability

Theorem 4.3. *A well formed, linearizable concurrent data structure, running on Montage, is buffered durably linearizable.*

Proof (sketch). We need to show that in any execution H containing a crash c, the state of the data structure after recovery reflects some consistent prefix of the linearized pre-crash history. Suppose that c occurs in epoch e of H. If $e \leq 2$, recovery will restore the initial state of the system, which reflects the null prefix of execution. If $e > 2$, Montage will discard all payloads created in epochs e and $e-1$, preserving those in existence as of a_{e-1}, and will pass these to the structure's recovery routine. This routine, by Constraint 6, will construct a new concrete state t such that $\mathcal{M}(t) = Q(r)$, where r is the set of payloads it was given. But r is precisely the set of payloads created by operations that linearized prior to a_{e-1}. If execution had reached quiescence immediately after those operations, Constraint 5 implies that the concrete state s of S would have been such that $\mathcal{M}(s) = Q(r)$. Thus the post-recovery state t reflects a consistent prefix of the linearized pre-crash history. □

4.3 Liveness

Theorem 4.4. *Montage is lock free during crash-free execution.*

Proof (sketch). The only loop in Montage lies within `BEGIN_OP`, where an update operation seeks to read the epoch clock and announce itself as active in that epoch, atomically. Each retry of the loop implies that the epoch has advanced. If we assume that the epoch advancing operation (which need not be nonblocking) always waits until at least one operation has completed in the old epoch, then an operation can be delayed in `BEGIN_OP` only if some other operation has completed. The `OldSeeNewException`, similarly, will arise (and cause some operations to start over) only if the epoch has advanced. □

5 IMPLEMENTATION DETAILS

Figure 3 shows pseudocode for Montage's core functionality. The "operation tracker" indicates, for each thread in the system, the epoch of its active operation (if any), together with lists of payloads to persist and free (reclaim) at future epoch boundaries. The lists are logically indexed by epoch, but only the most recent 2 or 3 are needed. For simplicity, Montage maintains four sets, and indexes into them using the 2 low-order bits of the epoch number. For convenience, each thread also caches the epoch of its currently active operation and last active operation (if any) in thread-local storage as `op_epoch` and `last_epoch`.

Aside from the epoch clock itself, payloads are the only data allocated in NVM. Each payload indicates the epoch in which it was created and whether it is new (`ALLOC`), a replacement of an existing payload (`UPDATE`), or an anti-payload (`DELETE`). `ALLOC` payloads are created in `PNEW`. `UPDATE` payloads are created in `set` (when the block being modified was created in an earlier epoch and cannot be updated in place). `DELETE` payloads (anti-payloads) are created in `PDELETE`; they live until the payload they are nullifying has been safely reclaimed, and are reclaimed in the following epoch to preserve the order of persistence.

```
Struct Payload
    enum type = {ALLOC, UPDATE, DELETE}
    uint64_t epoch
    uint64_t uid // shared between real and anti-payloads
Struct EpochSys
    // transient structures
    Tracker operation_tracker
    PBlk* to_persist[4] // recent 4 epochs
    PBlk* to_free[4] // recent 4 epochs
    operation_local uint64_t op_epoch
    operation_local uint64_t last_epoch
    // persistent structures
    uint64_t curr_epoch
    Function osn_check (Payload* p) : void
        if op_epoch < p→epoch then
            throw OldSeeNewException
    Function advance_epoch ( ) : void
        operation_tracker.wait_all(curr_epoch - 1)
        to_persist[(curr_epoch - 1) % 4].persist_all( )
        sfence
        curr_epoch.atomic_increment( )
EpochSys* Recoverable::esys
Macro BEGIN_OP : void
    repeat
        esys → op_epoch = esys → curr_epoch
        esys → operation_tracker.register(tid, esys → op_epoch)
    until esys→op_epoch == esys→curr_epoch
    forall e needs to be persisted for some sync() do
        to_persist[e % 4].persist_local(tid)
    if op_epoch > last_epoch then
        forall e between last_epoch-1 and
        min(last_epoch+1, op_epoch-2) do
            to_free[e % 4].free_local(tid)
            sfence
        last_epoch = op_epoch
Macro END_OP : void
    esys → op_epoch = NULL
    esys → operation_tracker.unregister(tid)
Function payload.get_x( ) : typeof(x)
    esys → osn_check(this)
    return this→ x
Macro PNEW (Type, ...) : Type*
    new_payload = new Type(...)
    new_payload→ epoch = esys → op_epoch
    new_payload→ type = ALLOC
    return new_payload
Macro PDELETE (Payload* p) : void
    esys → osn_check(p)
    if p.epoch == esys → op_epoch then
        if p→type == ALLOC then
            delete(p)
            return
        else
            p→ type = DELETE
    else
        anti_payload = new Payload( )
        anti_payload→ type = DELETE
        anti_payload→ uid = p→ uid
        esys → to_persist[esys → op_epoch % 4].add(anti_payload)
        esys → to_free[(esys → op_epoch + 1) % 4].add(anti_payload)
    esys → to_free[esys → op_epoch % 4].add(p)
Function payload.set_x (typeof(x) y) : Payload*
    esys → osn_check(this)
    if this→ epoch == esys → op_epoch then
        this→ x = y
        esys → to_persist[esys → op_epoch % 4].add(this)
        return this
    else // this→ epoch < esys → op_epoch
        new_payload = copy(this)
        new_payload→ epoch = esys → op_epoch
        new_payload→ type = UPDATE
        new_payload→ x = y
        esys → to_persist[esys → op_epoch % 4].add(new_payload)
        esys → to_free[esys → op_epoch % 4].add(this)
        return new_payload
```

Figure 3: Montage Pseudocode.

5.1 Storage Management

Space for payloads in Montage is managed by a variant of the Ralloc persistent allocator [3]. Ralloc is in turn based on the nonblocking allocator of Leite and Rocha [27]. Ralloc has very low overhead and excellent locality during crash-free operation. Almost all metadata is kept in transient memory, and most allocation and deallocation operations perform no write-back or fence instructions.

In its original form, Ralloc performs garbage collection after a crash to identify the blocks that are currently in use; all others are returned to the free list. For Montage, we modified the recovery mechanism to simply peruse all blocks, and to keep all and only those that are labeled as having been created at least two epochs ago. (These blocks will of course have been written back at some previous epoch boundary.) Montage passes the recovered blocks (i.e., payloads) to the application data structure, which is then responsible for rebuilding transient state. To facilitate parallel recovery, the application may request that the blocks be returned via k separate iterators, to be used by k separate application threads. As a point of reference, the recovery code for our Montage hashmap is less than 50 LOC.

5.2 Configuration Options

A wide variety of concrete designs could be used to flesh out the pseudocode of Figure 3. Natural questions include:

- Should the `advance_epoch` function be called periodically by application (worker) threads—e.g., from within the API calls—or should it be called by a background thread?
- Once `advance_epoch` has been called, should it be executed by a single thread, or should it be parallelized? (The Pronto system, a possible inspiration, can be configured to perform all writes-back on the sister hyperthread of the worker that wrote the data [32].)
- Is the answer to the previous question the same for both writes-back and storage reclamation? Perhaps some tasks are better performed on the cores where payloads or payload lists are likely to be in cache?
- Should all writes-back for a given epoch be delayed until the end, or does it make sense to start some of them earlier? One might, for example, employ a circular buffer in each worker, and issue writes-back one at a time, all at once, or perhaps half a buffer at a time, as the buffer fills.
- How long should an epoch be? Should it be measured in time, operations performed, or payloads written?

We performed a variety of experiments to evaluate the impact on performance of various answers to these questions; Figures 4 and 5 show some of the results. In each graph, the first four groups of bars use per-thread circular buffers of 2, 16, 64, or 256 payloads, respectively. When these buffers overflow, the oldest entries are written back incrementally. A single background thread serves to advance the epoch, at a frequency indicated by bars within each group. The background thread also writes back any remaining items in the per-worker-thread buffers at each epoch boundary, and performs all memory reclamation. In the fifth group of bars, reclamation is moved into the worker threads, which reclaim freed payloads and anti-payloads from epoch e at the beginning of epochs $e + 2$ and $e + 3$, respectively.

The final three groups of bars are provided for reference only (the final two do not correctly implement persistence). DirWB performs an immediate write-back after every update; Montage(T) places payloads in NVM but omits writes-back and reclaims deleted payloads immediately; Buf=64+DirFree buffers its writes-back but performs immediate reclamation.

While the best parameters depend to some degree on the nature of the application and the underlying hardware, we obtained good overall performance by using an epoch length of 10 ms, buffering up to 64 writes-back in each thread during each epoch (incrementally writing back any excess), and arranging for a single background thread to advance the epoch and perform remaining writes-back.

Parallel (incremental) write-back turns out to be essential: a single background thread is unable to keep up with more than a small number of worker threads in a high-throughput microbenchmark if it is responsible for all writes-back (drawn from unbounded buffers) at the end of every epoch. The background thread *does* seem to be able to keep up with reclamations, however; moving these into the worker threads has a small negative impact on throughput due to critical path dilation. A separate background thread for each worker, running on the worker's core, improves throughput in some but not all cases; in general this strikes us as a poor use of resources.

While the effect of epoch length depends on an application's cache footprint, it is generally smaller than the effect of the write-back buffer size. Further insight into the impact of shorter epochs can be found in Figure 9 (Sec. 6.1.2), in which we arrange for each thread of a hash table microbenchmark to invoke a `sync` operation every k operations, for various values of k. Throughput doesn't begin to drop off until the effective epoch length (the time between `sync` calls) is under 1 ms.

To minimize the latency of `sync` (not shown in Fig. 3), the caller helps perform the writes-back of its peers before updating the global epoch counter. At the beginning of each operation, a worker also helps to persist its payloads from the previous epoch if they are needed by any active `sync`. A variant of the *mindicator* of Liu et al. [30] keeps track, efficiently, of the oldest epoch for which unpersisted payloads still exist.

6 EXPERIMENTAL RESULTS

In this section we present a series of experiments that use microbenchmarks to compare the performance of Montage to that of competing systems (Sec. 6.1), validate the microbenchmark results with experiments using memcached (Sec. 6.2), demonstrate generality by persisting arbitrary graphs (Sec. 6.3), and assess the cost of recovery (Sec. 6.4).

All tests were conducted on a Linux 5.3.7 (Fedora 30) server with two Intel Xeon Gold 6230 processors, with 20 physical cores and 40 hyperthreads in each socket—a total of 80 hyperthreads. Threads in all experiments were pinned first one per core on socket 0, then on the extra hyperthreads of that socket, and then socket 1. Each socket has 6 channels of 128 GB Optane DIMMs and 6 channels of 32 GB DRAMs. We use ext4 to map NVM pages in direct access (DAX) mode. In all experiments, we allow Linux to allocate DRAM across the two sockets of the machine according to its default policy. The NVM is explicitly interleaved across sockets (`dm-stripe` with a 2 MB chunk size [40]). The source code of Montage is available at https://github.com/urcs-sync/Montage.

Systems and structures tested include the following:

Montage – as described in previous sections.
Friedman – the persistent lock-free queue of Friedman et al. [14].
Dalí – our reimplementation of the buffered durably linearizable hashmap of Nawab et al. [35].
SOFT – the lock-free hashmap of Zuriel et al. [50], which persists only semantic data but keeps a full copy in DRAM.
NVTraverse – a general transformation that converts transient "traversal data structures" into persistent ones. [13]
MOD – persistent structures (here, queues and hashmaps) as proposed by Haria et al. [17], who leverage history-preserving trees to linearize updates with a single write. The hashmap is implemented with per-bucket locking using MOD linked lists. This hashmap has lower time complexity and better scalability than the compressed hash-array mapped prefix-tree in the original MOD paper [17].
Pronto-Full and **Pronto-Sync** – the general-purpose system of Memaripour et al. [32], which logs high-level operation descriptions that can be replayed, starting from a checkpoint, to recover after a crash. We test both the synchronously logged and (on ≤ 40 threads) the "full" (asynchronous) version.
Mnemosyne – the general-purpose, pioneering system of Volos et al. [44], which adds persistence to the TinySTM transactional memory system [39].

For comparison purposes, we also include:

DRAM (T) and **NVM (T)** – high quality transient data structures built on DRAM and NVM, respectively, with no persistence support.
Montage (T) – a variant of Montage that still places payloads in NVM, but elides all persistence operations (no buffering, write-back instructions, delayed deletion, or epoch advance).

6.1 Microbenchmark Throughput

We have benchmarked Montage against the data structures and systems listed above, using queue and hashmap structures. Results appear in Figures 6 and 7. The Montage queue employs a single lock. The Montage hashmap has a lock per bucket—like all other competitors, it represents each bucket as a linked list. In work not reported here, we have developed nonblocking linked lists, queues, and maps, and various tree-based maps. In Section 6.3 we describe the implementation of a general graph, with operations to add and remove both vertices and edges.

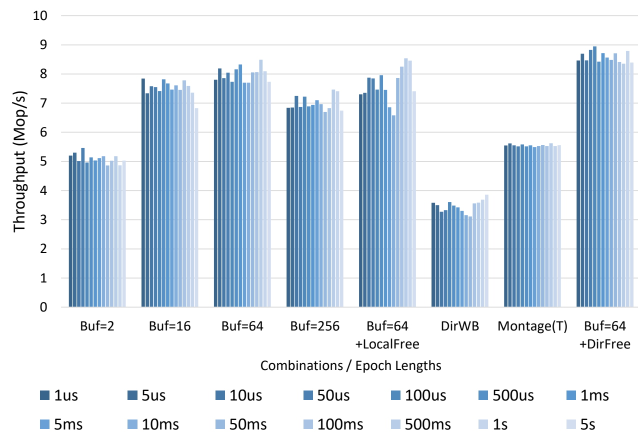


Figure 4: Design exploration on 40-thread hash table

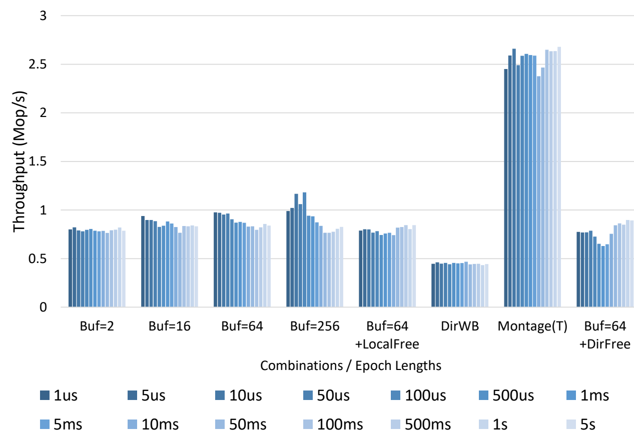


Figure 5: Design exploration on 1-thread queue

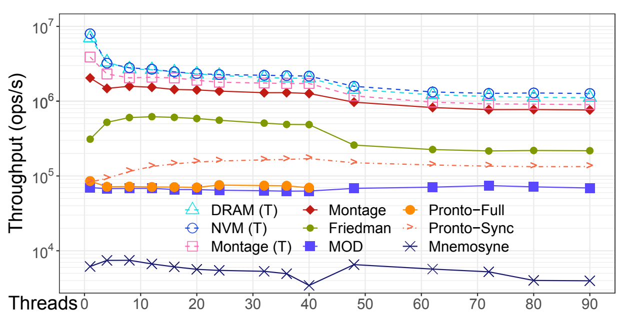


Figure 6: Throughput of concurrent queues.

The queue microbenchmark runs a 1:1 enqueue:dequeue workload. For the map we report both write-dominant (0:1:1 get:insert:remove) and read-dominant (18:1:1 get:insert:remove) results, with 0.5 million elements preloaded in 1 million hash buckets. The value size in queues and maps is 1 KB. Key values range from 1 to 1 million, converted to a string and padded to 32 B. Each workload runs for 30 seconds. Results were averaged over 3 trials for each data point. Since SOFT does not support atomic `update`s for existing keys, our benchmark does not include these. Separate experiments (not shown) confirm that the use of `update` does not significantly alter the curves of other systems.

As shown in Figures 6 and 7, Montage data structures generally perform as fast as transient structures running on NVM (they may even outperform NVM (T), given transient indexing in DRAM). Compared to DRAM (T), Montage adds as little as 30% overhead in queues, and less than 65% on the highly concurrent hashmap in most cases. With the exception of SOFT, Montage also outperforms all tested persistence systems on all four workloads. The Montage queue provides up to 6× the throughput of Friedman et al.'s queue, and is one to two orders of magnitude faster than the MOD, Pronto, and Mnemosyne queues. For hashmaps, Montage runs up to 4× faster than MOD, 4×–30× faster than Dalí, NVTraverse and Pronto, and nearly two orders of magnitude faster than Mnemosyne on the write-dominant workloads. On the read-dominant workload, Montage still has up to 4× the throughput of MOD, the fastest general-purpose competitor system.

The exceptional case is SOFT, which maintains—and reads from—a full copy of the data in DRAM, and which works for sets and mappings only, without atomic `update`. Nonetheless, Montage is close to or outperforms SOFT at low thread counts and on the read-dominant workload, and still achieves more than one-third the throughput of SOFT at high thread counts. Interestingly, Montage and NVM (T) stop scaling at 12 and 20 threads on the write-dominant workload; this may reflect multithreading contention in Intel's NVM write combining buffer and write pending queues. Similar contention may also explain why NVTraverse, which has writes-back and a fence in both read and write operations, is able to keep up with Montage's performance at lower thread counts, but subsequently falls behind.

It may seem surprising that NVM (T) has higher throughput than DRAM on queue benchmarks. This is because NVM (T) uses Ralloc instead of jemalloc; we believe Ralloc's block layout provides enqueue/dequeue workloads with better cache locality.

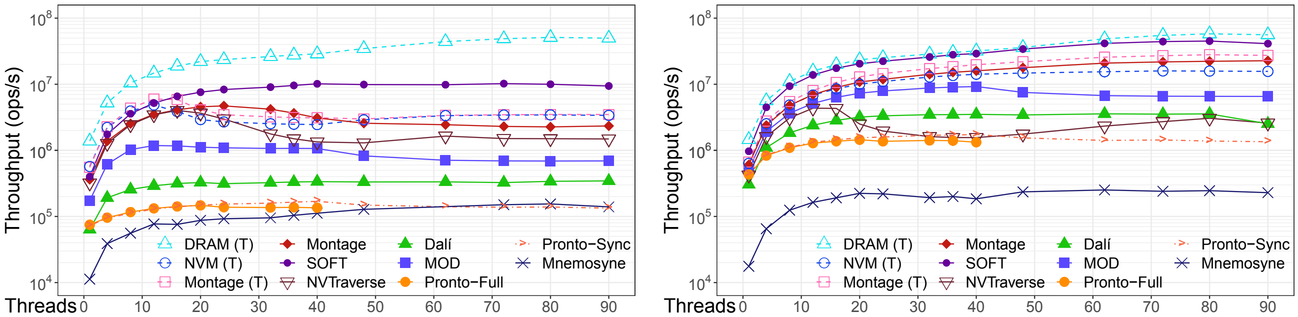
6.1.1 *Payload Size.*

To assess the impact of operation footprint on relative performance, we repeated our queue and read-write hashmap experiments with a single thread but with payloads varying from 16 B to 4 KB. Results appear in Figure 8 (here with a mixed read-write workload for the hashmap).

At all payload sizes, Montage continues to outperform all persistent competitors other than SOFT. Interestingly, in write-dominant hashmap experiments (not shown), the SOFT curve drops more sharply than the Montage curve, and crosses over at just 256 B: the overhead of (strict) durable linearizability increases with larger payloads, while Montage benefits more from its buffering.

6.1.2 *Sync Frequency.*

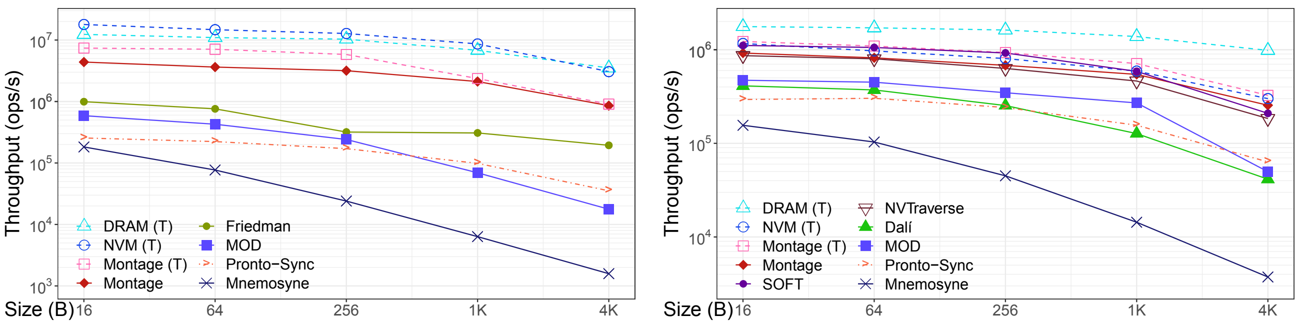
As noted in Section 1, buffered durable linearizability mirrors the behavior of traditional file and database systems: operations are permitted to return before they reach persistence. An application that must be *certain* of persistence (e.g., before sending confirmation to a remote client over the internet) can call a Montage `sync` operation. In the extreme, an application can obtain strict durable linearizability by calling `sync` after every operation, but this will reduce performance (much as it does for traditional block devices) and is generally overkill.



(a) 0:1:1 get:insert:remove

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

Figure 7: Throughput of concurrent hashmaps.



(a) Single-threaded Queues

(b) Single-threaded Hashmap—2:1:1 get:insert:remove

Figure 8: Throughput of single-threaded data structures (log-scale x-axis).

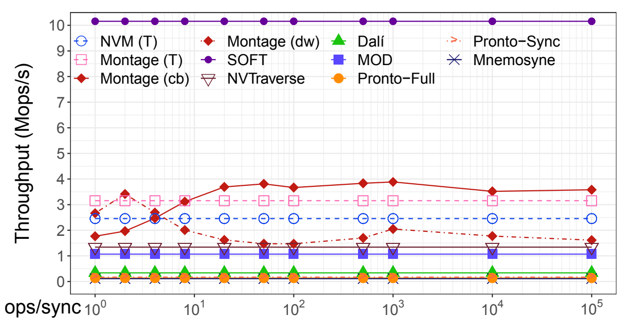


Figure 9: Throughput of 40-thread hashmaps with a sync every x operations on average (log-scale x-axis; linear y-axis).

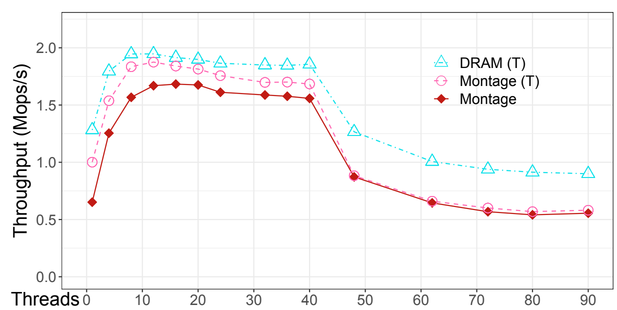


Figure 10: memcached throughput on YCSB-A (linear y-axis).

To assess the impact of sync, we repeated our write-dominant hashmap experiments with 40 threads but with calls to sync interspersed in every thread every 1 to 10^5 operations on average. In Figure 9, we employ two different write-back strategies in Montage: Montage (dw) writes back and flushes all written payloads at the end of each operation; Montage (cb) tracks updates in 64-entry per-thread buffers, as described in Section 5.2. With one sync every 40 operations or fewer ($\geq$ 2500 syncs per thread per second), Montage (cb) suffers from bookkeeping overhead on epoch advances; it wins with less frequent sync calls. Significantly, with either configuration, Montage outperforms NVTraverse, MOD, and Pronto even with a sync after *every* operation.

6.2 Hashmap Validation Using memcached

To confirm our data structure results in a more realistic setting, we use Montage to persist a variant of *memcached* developed by Kjellqvist et al. [23]. This variant links directly to a multithreaded client application, dispensing with the usual socket-based communication. It was appealing for our experiments because the authors had already converted it to use Ralloc instead of the benchmark's usual custom allocator.

Figure 10 compares the resulting (fully persistent, recoverable) version of memcached to the transient version of Kjellqvist et al., placing items in DRAM or in NVM; since the index always stays in DRAM, the latter is effectively equivalent to the configuration of Montage (T). Here the YCSB-A workload [10], with 1 M

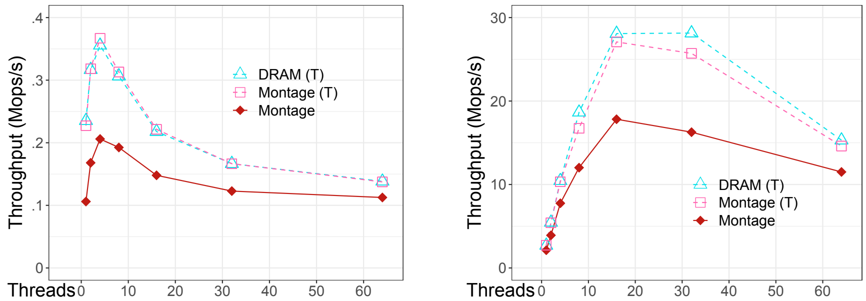


Figure 11: Graph microbenchmark throughput (linear y-axis) (AddEdge+RemoveEdge):(AddVertex+RemoveVertex) = 4:1 (left), 499:1 (right).

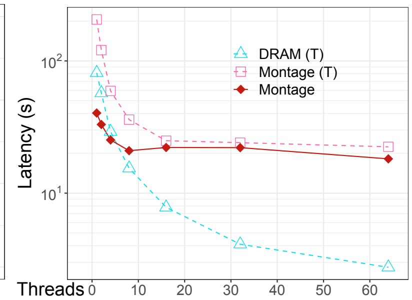


Figure 12: Time to rebuild Orkut graph.

records, comprises 2.5 M read and 2.5 M update operations, evenly distributed across threads. Data points reflect the average of three trials. As in the microbenchmark results, Montage performs within a small constant factor of purely transient structures.

6.3 Generality in Graphs

As noted in Section 3.1, a Montage programmer must avoid long chains of pointers. In a persistent graph, we therefore arrange for edge payloads to point to their endpoint vertices, but not vice versa. A more conventional representation of connectivity is then kept in transient memory, with the (typically large) edge and vertex attributes appearing only in payloads. We regard this representation as a strong indication of Montage's generality. Using it, we compare performance (as in the memcached experiments) to transient graphs placed entirely in DRAM or partially in NVM. Figure 11 shows results for a microbenchmark that performs a mix of `AddEdge`, `RemoveEdge`, `AddVertex`, and `RemoveVertex` operations. The first two of these take vertex IDs as source and destination. `AddVertex` connects a new vertex to (on average) 32 other vertices; `RemoveVertex` clears all adjacent edges. We keep identifiers in each vertex payload, and name them in edge payloads. `AddEdge` and `RemoveEdge` do not affect any vertex payload; `RemoveVertex` deletes all edge payloads that name the deleted vertex.

To initialize the graph, we add $10^6/2$ vertices out of the total capacity of 10^6. For each initial vertex (as in `AddVertex`) we randomly create 32 edges to other vertices. While benchmarking, we vary the portion of edge and vertex operations (4:1 and 499:1 in our experiments), and carefully distribute to `Add` and `Remove` operations so that the number of existing vertices and the average vertex degree remain statistically stable. Each workload runs for 30 seconds. Results were averaged over 3 trials for each data point. The persistent Montage graph performs within a factor of 2 of the fully transient graph, mirroring the results of previous sections and confirming Montage's utility for arbitrary linked structures. While the average vertex degree is modest in these experiments, `AddVertex` and `RemoveVertex` operations are still somewhat expensive in both the persistent and transient case. When these operations are called less often (right half of Fig. 11), overall throughput is higher.

6.4 Recovery Time

To assess the overhead of recovery in Montage, we measured both hash map and graph examples. In the hashmap case, we initialized the table with 2–64 million 1 KB elements, leading to a total payload size of 1–32 GB. With 1 recovery thread, Montage recovers the 1 GB data set in 0.7 s and the 32 GB data set in 41.9 s. With 8 recovery threads, it takes 0.4 and 13.8 s, respectively. Improving the scalability of recovery is a topic for future work.

As a second example, we compared the recovery time of a large Montage graph (the SNAP Orkut dataset [28, 48], a social network of ~3 M vertices and 117 M edges) to the time required to construct the same graph, in parallel, from a set of adjacency lists. The dataset is partitioned into many files, each of which uses a custom binary format that eliminates the need for string manipulation. Montage recovery is handled much like parallel construction: vertices and edges are added back to the graph in parallel. Because recovery is an internal graph operation, however, much of the locking can be elided by cyclically distributing vertices among threads, each of which creates a set of edge buffers to pass to other threads. Figure 12 demonstrates that recovery is even faster than construction on DRAM at low thread counts, and takes roughly as long as construction on NVM after 16 threads. Crucially, the Montage implementation has the advantage of supporting small changes to the graph without the need orchestrate persistence via file I/O.

7 CONCLUSIONS

We have introduced Montage, the first general-purpose system for buffered durable linearizability of persistent data structures. In comparison to systems that are (strictly) durably linearizable, Montage moves write-back and, crucially, fencing off the critical path of the application. Montage is built on top of the Ralloc nonblocking persistent allocator [3], which avoids both writes-back and fences in most allocation and deallocation operations. Nonblocking data structures remain nonblocking when implemented on top of Montage, though preempted threads can stall the advance of the persistence frontier.

Experiments with multiple data structures—including the hashmap of memcached—confirm that Montage dramatically outperforms prior general-purpose systems for persistence. It also outperforms—or is competitive with—existing special-purpose persistent data structures. In many cases, in fact, it rivals the performance of traditional transient data structures configured to use NVM instead of DRAM. This is generally the best performance one could hope for.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1717712, CNS-1900803, and CNS-1955498, by a Google Faculty Research award, and by a US Department of Energy Computational Science Graduate Fellowship (grant DE-SC0020347).

REFERENCES

[1] David Aksun and James Larus. 2021. Durability Through NVM Checkpointing (poster). In *12th Non-Volatile Memories Wkshp.*

[2] H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott. 2020. Nonblocking Persistent Software Transactional Memory. In *27th Intl. Conf. on High Performance Computing, Data, and Analytics (HiPC).*

[3] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *19th Intl. Symp. on Memory Management (ISMM).*

[4] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *ACM Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA).*

[5] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-ahead system for In-memory Non-volatile Data-structures. *Proc. of the VLDB Endowment* 8, 5 (Jan. 2015).

[6] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proc. of the VLDB Endowment* 8, 7 (Feb. 2015).

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. 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).* 14 pages. https://doi.org/10.1145/1950365.1950380

[8] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *24th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 14 pages. https://doi.org/10.1145/3297858.3304046

[9] Nachshon Cohen, David T. Aksun, and James R. Larus. 2018. Object-Oriented Recovery for Non-Volatile Memory. *Proc. of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018).

[10] Brian Cooper. 2010. YCSB Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[11] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *30th ACM Symp. on Parallel Algorithms and Architectures (SPAA).*

[12] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *Usenix Annual Technical Conf. (ATC).* https://www.usenix.org/conference/atc18/presentation/david

[13] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination Is More Important Than the Journey. In *41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).*

[14] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-free Queue for Non-volatile Memory. In *23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP).* 13 pages.

[15] Ellis R. Giles, Kshitij Doshi, and Peter Varman. 2015. SoftWrAP: A Lightweight Framework for Transactional Support of Storage Class Memory. In *31st Symp. on Mass Storage Systems and Technologies (MSST).*

[16] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Usenix Annual Technical Conf. (ATC).*

[17] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *25th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[18] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical persistence for multi-threaded applications. In *12th European Conf. on Computer Systems (EuroSys).*

[19] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th Usenix Conf. on File and Storage Technologies (FAST).*

[20] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[21] J. Izraelevitz, H. Mendes, and M. L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Intl. Symp. on Distributed Computing (DISC).*

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714v3.

[23] Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Safe, Fast Sharing of Memcached as a Protected Library. In *49th Intl. Conf. on Parallel Processing (ICPP).* Article 6, 8 pages. https://doi.org/10.1145/3404397.3404443

[24] R Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable transactional memory can scale with TimeStone. In *25th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[25] Harris Timothy L., Fraser Keir, and Pratt Ian A. 2002. A Practical Multi-word Compare-and-Swap Operation. In *16th Intl. Symp. on Distributed Computing (DISC).*

[26] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th Usenix Conf. on File and Storage Technologies (FAST).*

[27] Ricardo Leite and Ricardo Rocha. 2018. LRMalloc: A Modern and Competitive Lock-Free Dynamic Memory Allocator. In *13th Intl. Meeting on High Performance Computing for Computational Science (VECPAR).*

[28] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. on Intelligent Systems and Technology* 8, 1 (July 2016).

[29] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *51st Intl. Symp. on Microarchitecture (MICRO).*

[30] Yujie Liu, Victor Luchangco, and Michael Spear. 2013. Mindicators: A scalable approach to quiescence. In *33rd IEEE Intl. Conf. on Distributed Computing Systems (ICDCS).*

[31] Pratyush Mahapatra, Mark D Hill, and Michael M Swift. 2019. Don't Persist All: Efficient Persistent Data Structures. arXiv:1905.13011.

[32] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *25th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[33] Amirsaman Memaripour and Steven Swanson. 2018. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *36th Intl. Conf. on Computer Design (ICCD).*

[34] Moohyeon Nam, Hokeun Cha, Kibeom Jin, Jiwon Seo, and Beomseok Nam. 2020. B3-tree: Byte-Addressable Binary B-Tree for Persistent Memory. *ACM Trans. on Storage* 16, 3 (July 2020).

[35] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *Intl. Symp. on Distributed Computing (DISC).*

[36] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Intl. Conf. on the Management of Data (SIGMOD).*

[37] Matej Pavlovic, Alex Kogan, Virendra J Marathe, and Tim Harris. 2018. Brief announcement: Persistent multi-word compare-and-swap. In *ACM Symp. on Principles of Distributed Computing (PODC).*

[38] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *49th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN).*

[39] Torvald Riegel, Pascal Felber, and Christof Fetzer. 2006. A Lazy Snapshot Algorithm with Eager Validation. In *20th Intl. Symp. on Distributed Computing (DISC).*

[40] Steve Scargall. 2018. Using Persistent Memory Devices with the Linux Device Mapper. https://pmem.io/2018/05/15/using_persistent_memory_devices_with_the_linux_device_mapper.html.

[41] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *3rd VLDB Wkshp. on In-Memory Data Management and Analytics (IMDM).*

[42] Usharani U. and Andy M. Rudoff. 2017. Introduction to Programming with Persistent Memory from Intel. https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel.

[43] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *9th Usenix Conf. on File and Storage Technologies (FAST).*

[44] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[45] Chundong Wang, Qingsong Wei, Lingkun Wu, Sibo Wang, Cheng Chen, Xiaokui Xiao, Jun Yang, Mingdi Xue, and Yechao Yang. 2018. Persisting RB-Tree into NVM in a Consistency Perspective. *ACM Trans. on Storage* 14, 1 (Feb. 2018).

[46] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent memory threads harnessing versioned shadow copies. In *41st ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).*

[47] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-execute More. In *26th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[48] Jaewon Yang and Jure Leskovec. 2012. SNAP Dataset: Orkut Social Network and Ground-Truth Communities. https://snap.stanford.edu/data/com-Orkut.html.

[49] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th Usenix Conf. on File and Storage Technologies (FAST).*

[50] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-free Durable Sets. *Proc. of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019).