Transactional Composition of Nonblocking Data Structures

Wentao Cai[†]
wcai6@cs.rochester.edu
Computer Science Department
University of Rochester
Rochester, NY, USA

Haosen Wen[‡]
hwen5@cs.rochester.com
Computer Science Department
University of Rochester
Rochester, NY, USA

Michael L. Scott
scott@cs.rochester.com
Computer Science Department
University of Rochester
Rochester, NY, USA

ABSTRACT

This paper introduces *nonblocking transaction composition* (NBTC), a new methodology for atomic composition of nonblocking operations on concurrent data structures. Unlike previous software transactional memory (STM) approaches, NBTC leverages the linearizability of existing nonblocking structures, reducing the number of memory accesses that must be executed together, atomically, to only one per operation in most cases (these are typically the linearizing instructions of the constituent operations).

Our obstruction-free implementation of NBTC, which we call *Medley*, makes it easy to transform most nonblocking data structures into transactional counterparts while preserving their liveness and high concurrency. In our experiments, Medley outperforms Lock-Free Transactional Transform (LFTT), the fastest prior competing methodology, by 40–170%. The marginal overhead of Medley's transactional composition, relative to separate operations performed in succession, is roughly 2.2×.

For *persistent* data structures, we observe that failure atomicity for transactions can be achieved "almost for free" with epoch-based *periodic persistence*. Toward that end, we integrate Medley with *nbMontage*, a general system for periodically persistent data structures. The resulting *txMontage* provides ACID transactions and achieves throughput up to two orders of magnitude higher than that of the OneFile persistent STM system.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent algorithms**; • **Theory of computation** → **Parallel computing models**; • **Hardware** → *Non-volatile memory*.

KEYWORDS

nonblocking data structures, transactions, persistent memory

ACM Reference Format:
Wentao Cai, Haosen Wen, and Michael L. Scott. 2023. Transactional Composition of Nonblocking Data Structures. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3558481.3591079

SPAA '23, June 17–19, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9545-8/23/06…$15.00
https://doi.org/10.1145/3558481.3591079

1 INTRODUCTION

Nonblocking concurrent data structures, first explored in the 1970s, remain an active topic of research today. In such structures, there is no reachable state of the system that can prevent an individual operation from making forward progress. This *liveness* property is highly desirable in multi-threaded programs that aim for high scalability and are sensitive to high tail latency caused by inopportune preemption of resource-holding threads.

Many multi-threaded systems, including those for finance, travel [30], warehouse management [6], and databases in general [39], need to compose operations into transactions that occur in an all-or-nothing fashion (i.e., atomically). Concurrent data structures, however, ensure atomicity only for individual operations; composing a transaction across operations requires nontrivial programming effort and introduces high overhead. Preserving nonblocking liveness for every transaction is even more difficult.

One potential solution can be found in *software transactional memory* (STM) systems, which convert almost arbitrary sequential code into speculative transactions. Several STM systems provide nonblocking progress [10, 19, 25, 26, 37]. Most instrument each memory access and arrange to restart operations that conflict at the level of individual loads and stores. The resulting programming model is attractive, but the instrumentation typically imposes 3–10× overhead [34, Sec. 9.2.3].

Inspired by STM, Spiegelman et al. [36] proposed *transactional data structure libraries* (TDSL), which introduce (blocking) transactions for certain hand-modified concurrent data structures. By observing that reads need to be tracked only on *critical nodes* whose updates may indicate semantic conflicts, TDSL reduces read set size and achieves better performance than general STMs.

Herlihy and Koskinen [18] proposed *transactional boosting*, a (blocking) methodology that allows an STM system to incorporate operations on existing concurrent data structures. Using a system of *semantic locks* (e.g., with one lock per key in a mapping), transactions arrange to execute concurrently so long as their boosted operations are logically independent, regardless of low-level conflicts. A transaction that restarts due to a semantic conflict (or to a low-level conflict outside the boosted code) will roll back any already-completed boosted operations by performing explicitly identified *inverse operations*. An insert(k,v) operation, for example, would be rolled back by performing remove(k). Transactional boosting leverages the potential for high concurrency in existing data structures, but is intrinsically lock-based, and is not fully general:

[†]Now at Google. [‡] Now at Huawei Canada.
This work was supported in part by NSF grants CCF-1717712, CNS-1900803, and CNS-1955498.

operations on a single-linked FIFO queue, for example, have no obvious inverse.

In work concurrent to TDSL, Zhang et al. [43] proposed the *Lock-Free Transactional Transform* (LFTT), a nonblocking methodology to compose nonblocking data structures, based on the observation that only certain nodes—those critical to transaction semantics—really matter in conflict management. Each operation on an LFTT structure *publishes*, on every critical node, a description of the transaction of which it is a part, so that conflicting transactions can see and *help* each other. A remove(7) operation, for example, would publish a description of its transaction on the node in its structure with key 7. Initially, LFTT supported only static transactions, whose constituent operations were all known in advance. Subsequently, LaBorde et al. [23] proposed a *Dynamic Transactional Transform* (DTT) that generalizes LFTT to dynamic transactions (specified as lambda expressions). Concurrently, Elizarov et al. [8] proposed LOFT, which is similar to LFTT but avoids incorrectly repeated helping.

Unfortunately, as in transactional boosting, the need to identify critical nodes tends to limit LFTT and DTT to data structures representing sets and mappings. DTT's publishing and helping mechanisms also require that the "glue" code between operations be fully reentrant (to admit concurrent execution by helping threads [23]) and may result in redundant work when conflicts arise. Worse, for read-heavy workloads, LFTT and DTT require readers to be *visible* to writers, introducing metadata updates that significantly increase contention in the cache coherence protocol.

In our work, we propose *NonBlocking Transaction Composition* (NBTC), a new methodology that can create transactional versions of a wide variety of concurrent data structures while preserving nonblocking progress and incurring significantly lower overhead than traditional STM. The intuition behind NBTC is that in already nonblocking structures, only *critical memory accesses*—for the most part, the linearizing load and compare-and-swap (CAS) instructions—need to occur atomically, while most pre-linearization memory accesses can safely be executed as they are encountered, and post-linearization accesses can be postponed until after the transaction commits.

In comparison to STM, NBTC significantly reduces the number of memory accesses that must be instrumented—typically to only one per constituent operation. Unlike transactional boosting and transactional transforms, NBTC brings the focus back from semantics to low-level memory accesses, thereby enabling mechanical transformation of existing structures and accommodating almost arbitrary abstractions—much more than sets and mappings. NBTC also supports dynamic transactions, invisible readers, and non-reentrant "glue" code between the operations of a transaction. The one requirement for compatibility is that the linearization points of constituent operations must be *immediately identifiable*: each update operation must be able to tell when it has linearized at run time, without performing any additional shared-memory accesses. Most nonblocking structures in the literature appear to meet this requirement.

To assess the practicality of NBTC, we have built an obstruction-free implementation, *Medley*, that uses a variant of Harris et al.'s multi-word CAS [16] to execute the critical memory accesses of each transaction atomically, eagerly resolving conflicting transactions as they are discovered. Using Medley, we have created NBTC versions of Michael and Scott's queue [29], Fraser's skiplist [10], the rotating skiplist of Dick et al. [7], Michael's chained hash table [28], and Natarajan and Mittal's binary search tree [31]. All of the transformations were straightforward.

In the traditional language of database transactions [15], Medley provides isolation and consistency. Building on recent work on persistent memory, we have also integrated Medley with our *nbMontage* system [2] to create a system, *txMontage*, that provides failure atomicity and durability as well—i.e., full ACID transactions. Specifically, we leverage the *epoch* system of nbMontage, which divides time into coarse-grain temporal intervals and recovers, on failure, to the state of a recent epoch boundary. By folding a check of the epoch number into its multi-word CAS, txMontage ensures that operations of the same transaction always linearize in the same epoch, thereby obtaining failure atomicity and durability "almost for free."

Summarizing contributions:

- (Section 2) We introduce *nonblocking transaction composition* (NBTC), a new methodology with which to compose the operations of nonblocking data structures.
- (Section 3) Deploying NBTC, we implement *Medley*, a general system for transactional nonblocking structures. Medley's easy-to-use API and mechanical transform make it easy to convert compatible nonblocking structures to transactional form.
- (Section 4) We integrate Medley with nbMontage to create *txMontage*, providing not only transactional isolation and consistency, but also failure atomicity and durability.
- (Section 5) We argue that using Medley, transactions composed of nonblocking structures are nonblocking and strictly serializable. We also argue that transactions with txMontage provide a persistent variant of strict serializability analogous to the buffered durable linearizability of Izraelevitz et al. [21].
- (Section 6) We present performance results, confirming that Medley imposes relatively modest overhead and scales to large numbers of threads. Specifically, Medley outperforms LFTT by 1.4× to 2.7× and outperforms TDSL and the OneFile nonblocking STM [33] system by an order of magnitude. On persistent memory, txMontage outperforms nonblocking persistent STM by two orders of magnitude.

2 NONBLOCKING TRANSACTION COMPOSITION

Nonblocking transaction composition (NBTC) is a new methodology that fully leverages the linearizability of nonblocking data structure operations. NBTC obtains strict serializability by atomically performing only the *critical memory accesses* of composed operations. It supports a large subset of the nonblocking data structures in the literature (characterized more precisely below), preserving the high concurrency and nonblocking liveness of the transformed structures.

2.1 NBTC Composability

The key to NBTC composability is the *immediately identifiable linearization point*. Specifically:

Definition 2.1. A data structure operation has an *immediately identifiable linearization point* if:

(1) statically, we can identify every instruction that may potentially serve as the operation's linearization point. Such an instruction must be a load (for a read-only operation) or a compare-and-swap (CAS);
(2) dynamically, after executing a potentially linearizing instruction, we can determine whether it was indeed the linearization point. A linearizing load must be identified *before the operation returns*; a potentially linearizing CAS will *be* the linearization point if and only if it succeeds.

An operation can be determined to be read-only during its execution, typically after checking the return value of a load or failed CAS. Unlike an immediately identifiable linearizing CAS, a linearizing load can be identified retroactively, as late as the point at which the operation returns (more on this at the end of Section 2.2). There can be more than one potential linearization point in the code of an operation, but only one of them will constitute the linearization point in any given invocation.

Definition 2.2. A nonblocking data structure is *NBTC-composable* if each of its operations has an immediately identifiable linearization point.

While it may be possible to relax this definition, the current version accommodates a very large number of existing nonblocking structures.

2.2 The Methodology

It is widely understood that most nonblocking operations comprise a "planning" phase and a "cleanup" phase, separated by a linearizing instruction [11, 38]. Executing the planning phase does not commit the operation to success; cleanup, if needed, can be performed by any thread. The basic strategy in NBTC is to perform the planning for all constituent operations of the current transaction, then linearize all those operations together, atomically, and finally perform all cleanup. Our survey of existing data structures and composition patterns reveals two principle complications with this strategy.

The first complication involves the notion of a *publication point*, where an operation may become visible to other threads but not yet linearize. Because publication can alter the behavior of other threads, it must generally (like a linearization point) remain speculative until the entire transaction is ready to commit. An example can be seen in the binary search tree of Natarajan and Mittal [31], where an update operation o may perform a CAS that publishes its intent to linearize soon but not quite yet. After this publication point, either o itself or any other update that encounters the publication notice may attempt to linearize o (in the interest of performance, a read operation will ignore it). Notably, CAS instructions that serve to help other (already linearized) operations, without revealing the nature of the current operation, need not count as publication.

The second complication arises when a transaction, t, performs two or more operations on the same data structure and one of the later operations (call it o_2) depends on the outcome of an earlier operation (call it o_1). Here the thread executing t must proceed as if o_1 has completed. If o_1 requires cleanup (something that NBTC will normally delay until after transaction commit), o_2 may need to help o_1 before it can proceed, while other threads and transactions should not even be aware of o_1's existence.

Both complicating cases can be handled by introducing the notion of a *speculation interval* in which CAS instructions must be completed together for an operation to take effect as part of a transaction. This is similar to the *CAS executor* phase in a *normalized* nonblocking data structure [38], but not the same, largely due to the second complication. For an operation that becomes visible before its linearization point, it suffices to include in the speculation interval all CAS operations between the publication and linearization points, inclusive. For an operation o_2 that needs to see an earlier operation o_1 in the same transaction, it suffices to track the transaction's writes and to start o_2's speculation interval no later than the first instruction that accesses a location written by o_1.

Definition 2.3. A bit more precisely, we say:

- A CAS instruction in operation o of thread t in history H is *benign* if there is no extension H' of H such that t executes no more instructions in H' and yet o linearizes in H' nonetheless.
- The first CAS instruction of o that is not benign is o's *publication point* (this will often be the same as its linearization point).
- The *speculation interval* of o begins either at the publication point or at the first instruction that sees a value speculatively written by some earlier operation in the same transaction (whichever comes first) and extends through o's linearization point.
- A CAS in an update operation is *critical* if it lies in the speculation interval. A load in a read-only operation is critical if it is the immediately identifiable linearization point of the operation.

Without loss of generality, we assume that all updates to shared memory (other than initialization of objects not yet visible to other threads) are effected via CAS.

Given these definitions, the NBTC methodology is straightforward: To atomically execute a set of operations on NBTC-composable data structures, we transform every operation such that (1) instructions prior to the speculation interval and non-critical instructions in the speculation interval are executed on the fly as a transaction encounters them; (2) critical instructions are executed in a speculative fashion, so they will take effect, atomically, only on transaction commit; and (3) instructions after the speculation interval are postponed until after the commit.

Immediately identifying a critical CAS by its return value is necessary, as we need to decide, in the moment, whether to execute it speculatively or "for real." A load, on the other hand, is always side-effect free, so identifying it retroactively, as late as its operation's return, never compromises the speculativity of the transaction as a whole.

3 THE MEDLEY SYSTEM

To illustrate NBTC, we have written a system, *Medley*, that (1) instruments critical instructions, executes them speculatively, and commits them atomically using *M-compare-N-swap*, our variant of the multi-word CAS of Harris et al. [16]; (2) identifies and eagerly

```
template <class T> class CASObj { // Augmented atomic object
  T nbtcLoad();
  bool nbtcCAS(T expected, T desired, bool linPt, bool pubPt);
  /* Regular atomic methods: */
  T load(); void store(T desired); bool CAS(T expected, T desired);
};
class Composable { // Base class of all transactional objects
  template <class T> void addToReadSet(CASObj<T>*,T); // Register load
  void addToCleanups(function); // Register post-critical work
  template <class T> T* tNew(...); // Create a new block
  template <class T> void tDelete(T*); // Delete a block
  template <class T> void tRetire(T*); // Epoch-based safe retire
  TxManager* mgr; // Tx metadata shared among Composables
  struct OpStarter { OpStarter(TxManager*); } // RAII op starter
};
class TxManager { // Manager shared among composable objects
  void txBegin(); // Start a transaction
  void txEnd(); // Try to commit the transaction
  void txAbort(); // Explicitly abort the transaction
  void validateReads(); // Optional validation for opacity
};
struct TransactionAborted : public std::exception{ };
```

Figure 1: C++ API of Medley for transaction composition.

resolves transaction conflicts; and (3) delays non-critical cleanup until transaction commit.

3.1 API

Figure 1 summarizes Medley's API. Using this API, we transform an NBTC-composable data structure into a transactional structure as follows:

(1) Derive the data structure class from Composable.
(2) Declare fields to which critical accesses may be made using the CASObj template. Replace loads and CASes to such fields with nbtcLoad and nbtcCAS, respectively.
(3) Invoke addToReadSet for the critical load in a read operation, recording the address and the loaded value.
(4) Register each operation's post-critical work via addToCleanups.
(5) In each operation, replace every new and delete with tNew and tDelete, and replace every retire (for safe memory reclamation—SMR) with tRetire.
(6) Declare an OpStarter object at the beginning of each operation.

CASObj<T> augments each CAS-able 64-bit word (e.g., atomic<Node*>) with additional metadata bits for speculation tracking (details in Section 3.2). It provides specialized load and CAS operations, as well as the usual methods of atomic<T>. To dynamically identify the speculation interval, nbtcCAS takes two extra arguments, linPt and pubPt, that indicate whether this call, should it succeed, will constitute its operation's linearization or/and publication point. In a similar vein, addToReadSet can be called after an nbtcLoad to indicate that this was (or is likely to have been) the linearizing load of a read-only operation, and should be tracked for validation at commit time.

Composable is a base class for transactional objects. It provides a variety of NBTC-related methods, including support for *safe memory reclamation* (SMR), used to ensure that nodes are not reclaimed until one can be certain that no references remain among the private variables of other threads. Our current implementation of SMR uses epoch-based reclamation [10, 17, 27]. For the sake of generality, Composable also provides an API for transactional boosting, which can be used to incorporate lock-based operations into Medley transactions (at the cost, of course, of nonblocking progress). We do not discuss this mechanism further in this paper.

The TxManager class manages transaction metadata and provides methods to initiate, abort, and complete a transaction. A TxManager instance is shared among all Composable instances intended for use in the same transactions. In each operation call, the manager distinguishes (via OpStarter) whether execution is currently inside or outside a transaction. If outside, all transactional instrumentation is elided; if inside, instrumentation proceeds as specified by the NBTC methodology.

Given that nonblocking operations can execute safely in any reachable state of the system, there is usually no need to stop the execution of a doomed-to-abort transaction as soon as a conflict arises—i.e., to guarantee *opacity* [14]. In exceptional cases (e.g., when later operations of a transaction cannot be called with certain combinations of parameters, or when aborts are likely enough that delaying them may compromise performance), the validateReads method can be used to determine whether previous reads remain correct.

To illustrate the use of Medley, Figure 2 highlights lines of code in Michael's nonblocking hash table [28] that must be modified for NBTC; Figure 3 then shows an example transaction that modifies two hash tables. In a real application, the catch block (written by the programmer) for TransactionAborted would typically loop back to the beginning of the transaction code to try again, possibly with additional code to avoid livelock (e.g., via backoff or hints to the underlying scheduler). In contrast to STM systems, Medley does not instrument the intra-transaction "glue" code between data structure operations. This "glue" code is always executed as regular code outside a transaction and is not managed by Medley; if it has side effects, the catch block for aborted transactions should compensate for these before the programmer chooses to retry or give up.

3.2 M-Compare-N-Swap

To execute the critical memory accesses of each transaction atomically, we employ a software-emulated *M-compare-N-swap* (MCNS) operation that builds on the double-compare-single-swap (RDCSS) and multi-word CAS (CASN) of Harris et al. [16]. Each transaction maintains a *descriptor* that contains a read set, a write set, and a 64-bit triple of thread ID, serial number, and status, as shown in Figure 4. Descriptors are pre-allocated on a per-thread basis within a TxManager instance, and are reused across transactions. A status can be InPrep (initial state), InProg (ready to commit), Committed (after validation succeeds when InProg), or Aborted (explicitly by another thread when InPrep or due to failed validation).

Each originally 64-bit word at which a critical memory access may occur is augmented with a 64-bit counter, together comprising a 128-bit CASObj. Each critical CAS installs a pointer to its descriptor in the CASObj and increments the counter; upon commit or abort, the descriptor is uninstalled and the counter incremented again. We leverage 128-bit CAS instructions on the x86 to change the original word and the counter together, atomically. The counter is odd when CASObj contains a pointer to a descriptor and even when it is a real value.

```
class MHashTable : public Composable {
struct Node { K key; V val; CASObj<Node*> next; };
// from p, find c >= k; nbtcLoad and tRetire may be used
bool find(CASObj<Node*>* &p, Node* &c, Node* &n, K k);
optional<V> get(K key) {
  OpStarter starter(mgr); CASObj<Node*>* prev = nullptr;
  Node *curr, *next; optional<V> res = {};
  if (find(prev,curr,next,key)) res = curr->val;
  addToReadSet(prev,curr);
  return res;
}
optional<V> put(K key, V val) { // insert or replace if key exists
  OpStarter starter(mgr);
  CASObj<Node*>* prev = nullptr; optional<V> res = {};
  Node *newNode = tNew<Node>(key, val), *curr, *next;
  while(true) {
    if (find(prev,curr,next,key)) { // update
      newNode->next.store(curr);
      if (curr->next.nbtcCAS(next,mark(newNode),true,true)) {
        res = curr->val;
        auto cleanup = [](){
          if (prev->CAS(curr,newNode)) tRetire(curr);
          else find(prev,curr,next,key);
        };
        addToCleanups(cleanup); // execute right away if not in tx
        break;
      }
    } else { // key does not exist; insert
      newNode->next.store(curr);
      if (prev->nbtcCAS(curr,newNode,true,true)) break;
    }
  }
  return res;
}};
```

Figure 2: Michael's lock-free hash table example (Medley-related parts highlighted).

```
void doTx(MHashTable* ht1, MHashTable* ht2, V v, K a1, K a2) {
  TxManager* mgr=ht1->mgr; assert(mgr==ht2->mgr);
  try { // transfer `v' from account `a1' in `ht1' to `a2' in `ht2'
    mgr->txBegin();
    V v1 = ht1->get(a1); V v2 = ht2->get(a2);
    if (!v1.hasValue() or v1.value() < v) mgr->txAbort();
    ht1->put(a1, v1.value() - v); ht2->put(a2, v + v2.valueOr(0));
    mgr->txEnd();
  } catch (TransactionAborted) { /* transaction aborted */ }
}
```

Figure 3: Transaction example on Michael's hash table.

```
struct Desc {
  map<CASObj* addr,{uint64 val,cnt}>* readSet;
  map<CASObj* addr,{uint64 oldVal,cnt,newVal}>* writeSet;
  atomic<uint64> status;//63..50 tid; 49..2 serialNumber; 1..0 status
  enum STATUS { InPrep=0, InProg=1, Committed=2, Aborted=3 };
};
struct CASObj { atomic<uint128> val_cnt; };
```

Figure 4: Descriptor and CASObj structures.

Each instance of MCNS proceeds through phases that install descriptors, finalize status, and uninstall descriptors. The first two phases are on the critical path of a data structure operation. A new transaction initializes metadata in its descriptor (at txBegin): it clears the read and write sets, increments the serial number, and resets the status to InPrep. The installing phase then occurs over the course of the transaction: Each critical load records its address, counter, and value in the read set. Each critical CAS records its address, old counter, old value, and desired new value in the write set; it then installs a pointer to the descriptor in the CASObj. Pseudocode for the installing phase appears in Figure 5.

```
1  void TxManager::txBegin() {
2    desc->readSet->clear(); desc->writeSet->clear();
3    status.store((status.load() & ~3) + 4);
4  }
5  T CASObj::nbtcLoad() {
6  retry:
7    {val,cnt} = val_cnt.load();
8    if (cnt % 2) { // is descriptor
9      if (val == desc) {
10       startSpeculativeInterval();
11       return desc->writeSet[this].newVal;
12     } else val->tryFinalize(this, {val,cnt});
13     goto retry; // until object has real value
14   }
15   ... /* Record `this' and `cnt' to be added to readSet */
16   return val;
17 }
18 void Composable::addToReadSet(CASObj<T>* obj, T val) {
19   ... /* Retrieve `cnt' by `obj` */
20   mgr->readSet[obj] = {val,cnt};
21 }
22 bool CASObj::nbtcCAS(T expected,T desired,bool linPt,bool pubPt){
23 retry:
24   {val,cnt} = val_cnt.load();
25   if (cnt % 2) { // is descriptor
26     if (val != desc) { // not own descriptor
27       val->tryFinalize(this, {val,cnt});
28       goto retry; // until object has real value
29     }
30     startSpeculativeInterval();
31   } else if (val != expected) return false;
32   if (pubPt) startSpeculativeInterval();
33   if (inSpeculativeInterval()) { // Is critical CAS
34     desc->writeSet[this] = {val,cnt,desired};
35     bool ret = true;
36     if (!(cnt % 2)) ret = this->CAS({val,cnt},{desc,cnt+1});
37     if (!ret) desc->writeSet.remove(this);
38     if (linPt and ret) endSpeculativeInterval();
39     return ret;
40   } else return CAS(expected, desired);
41 }
```

Figure 5: Pseudocode for installing phase of MCNS.

To spare the programmer the need to reason about counters, nbtcLoad makes a record of its ⟨counter, object⟩ pair (line 15 in Fig. 5); addToReadSet then adds this pair (and the specified CASObj) to the transaction's read set (line 20).

When a thread encounters its own descriptor, nbtcLoad returns the speculated value from the write set (line 11 in Fig. 5). Likewise, nbtcCAS updates the write entry (line 34). Such encounters automatically initiate the speculation interval (lines 10, 30, and 32), which then extends through the linearization point of the current operation (line 38).

If an operation encounters the descriptor of some other thread, it gets that descriptor out of the way by calling tryFinalize (Fig. 6). This method aborts the associated transaction if the descriptor is InPrep, helps complete the commit if InProg, and in all cases uninstalls the descriptor from the CASObj in which it was found. Similar actions occur when a thread is forced to abort or reaches the end of its transaction and attempts to commit (lines 39–58 in Fig. 6). Whether helping or acting on its own behalf, a thread performing an MCNS must verify that the descriptor is still responsible for the CASObj through which it was discovered (line 9) and (if committing) that the values in the read set are still valid (line 25). After CAS-ing the status to Committed or Aborted, the thread uninstalls the descriptor from

```
1  bool Desc::stsCAS(uint64 d, STATUS expected, STATUS desired) {
2    d = d & ~3; return status.CAS(d + expected, d + desired);
3  }
4  bool Desc::setReady(){return stsCAS(status.load(),InPrep,InProg);}
5  bool Desc::commit(uint64 d){return stsCAS(d,InProg,Committed);}
6  bool Desc::abort(uint64 d){return stsCAS(d,d & 1,Aborted);}
7  void Desc::tryFinalize(CASObj* obj, uint128 var) {
8    uint64 d = status.load();
9    if (obj->val_cnt.load() != var) // ensure d refers to right tx
10     return;
11   if (d & 3 == InPrep) {
12     abort(d);
13     uint64 newd = status.load();
14     if (newd & ~3 != d & ~3) return; // serial number mismatch
15     d = newd;
16   }
17   if (d & 3 == InProg) {
18     if (validateReads(d)) commit(d);
19     else abort(d);
20   }
21   uninstall(status.load());
22 }
23 bool Desc::validateReads() {
24   for (e:*readSet)
25     if ({e.val,e.cnt} != e.addr->load()) return false;
26   return true;
27 }
28 void Desc::uninstall(uint64 d) {
29   if (d & 3 == Committed)
30     for (e:*writeSet)
31       e.addr->CAS({this,e.cnt+1}, {e.newVal,e.cnt+2});
32   else // Aborted
33     for (e:*writeSet)
34       e.addr->CAS({this,e.cnt+1}, {e.oldVal,e.cnt+2});
35 }
36 struct TxManager {
37   threadLocal vector<Function> cleanups, allocs;
38   threadLocal Desc* desc;
39   void txAbort() {
40     uint64 d = desc->status.load();
41     desc->abort(d);
42     desc->uninstall(d);
43     for (f:allocs) f(); // undo tNew
44     throw TransactionAborted();
45   }
46   void txEnd() {
47     if (!desc->setReady()) txAbort();
48     else {
49       uint64 d = desc->status.load();
50       if (!desc->validateReads()) desc->abort(d);
51       else if (d & 3 == InProg) desc->commit(d);
52       d = desc->status.load();
53       if (d & 3 == Committed) {
54         desc->uninstall(d);
55         for (f:cleanups) f();
56       } else txAbort();
57     }
58   }
59 };
```

Figure 6: Pseudocode of methods that finalize transactions.

all associated CASObjs, replacing pointers to the descriptor with the appropriate updated values (lines 31 and 34). Once uninstalling is complete, the owner thread calls cleanup routines (line 55) for a commit or deallocates tNew-ed blocks (line 43) for an abort.

Our design adopts invisible readers and eager contention management for efficiency and simplicity. Eager contention management admits the possibility of livelock—transactions that repeatedly abort each other—and therefore guarantees only obstruction freedom. Lazy (commit-time) contention management along with some total order of descriptor installment might allow us to preserve lock freedom for structures that provide it [35], but would significantly complicate the tracking and retrieving of uncommitted changes, and would not address starvation, which may be a bigger problem than livelock in practice; we consider these implementation choices orthogonal to the effectiveness of the NBTC methodology, and defer them to future work.

4 PERSISTENT MEMORY

Transactions developed, historically, in the database community; transactional memory (TM) adapted them to in-memory structures in multithreaded programs. The advent of cheap, low-power, byte-addressable nonvolatile memory (NVM) presents the opportunity to merge these two historical threads in a way that ideally leverages NBTC. Specifically, where TM aims to convert sequential code to thread-safe parallel code, NBTC assumes—as in the database world—that we are already in possession of efficient thread-safe structures and we wish to combine their operations atomically and durably. Given this assumption, it seems appropriate (as described at the end of Sec. 3.1) to assume that the programmer is responsible for the "glue" code between operations, and to focus on the atomicity and durability of the composed operations.

4.1 Durable Linearizability

On machines with volatile caches, data structures in NVM will generally be consistent after a crash only if programs take pains to issue carefully chosen write-back and fence instructions. To characterize desired behavior, Izraelevitz et al. [21] introduced *durable linearizability* as a correctness criterion for persistent structures. A structure is durably linearizable if it is linearizable during crash-free execution and its long-term history remains linearizable when crash events are elided. Equivalently [12], each operation should persist between its invocation and response, and the order of persists should match the linearization order.

Many durably linearizable nonblocking data structures have been designed in recent years [3, 9, 12, 44]. Several groups have also proposed methodologies by which existing nonblocking structures can be made durably linearizable [11, 13, 21]. Other groups have developed persistent STM systems, but most of these have been lock-based [4, 5, 24, 40]. OneFile [33] and QSTM [1] are, to the best of our knowledge, the only nonblocking persistent STM systems. OneFile serializes transactions using a global sequence number, eliminating the need for a read set and improving read efficiency, but introducing the need for invasive data structure modifications and a 128-bit wide CAS. QSTM employs a global persistent queue for active transactions, avoiding the need for wide CAS and invasive structural changes, but with execution that remains inherently serial.

4.2 Lowering Persistence Overhead

Unfortunately, write-back and fence instructions tend to have high latency. Given the need for operations to persist before returning, durable linearizability appears to be intrinsically expensive. Immediate persistence for STM introduces additional overhead, as metadata for transaction concurrency control must also be eagerly written back and fenced.

To move high latency instructions off the application's critical path, Izraelevitz et al. [21] introduced the notion of *buffered* durable

linearizability (BDL). By allowing a modest suffix of pre-crash execution to be lost during post-crash recovery (so long as the overall history remains linearizable), BDL allows write-back and fence instructions to execute in batches, off the application's critical path. Applications that need to ensure persistence before communicating with the outside world can employ a sync operation, reminiscent of those in traditional file systems and databases.

First proposed in the context of the Dalí persistent hash table [32], *periodic persistence* was subsequently adopted by nbMontage [2], our general-purpose system to create BDL versions of existing nonblocking structures. The nbMontage system divides wall-clock time into "epochs" and persists operations in a batch at the end of each epoch. In the wake of a crash in epoch e, the system recovers all structures to their state as of the end of epoch $e-2$. To maximize throughput in the absence of crashes, nbMontage also distinguishes between data that are semantically significant (a.k.a. "payloads") and data that are merely performance enhancing (e.g., indices); the latter can be kept in DRAM and rebuilt during recovery. As an example, the payloads of a mapping are simply a pile of key-value pairs; the associated hash table, tree, or skiplist resides in transient DRAM. The payloads of a queue are ⟨serial number, item⟩ pairs.

To ensure that post-crash recovery always reflects a consistent state of each structure, every nbMontage operation is forced to linearize in the epoch with which its payloads have been labeled. Operations that take "too long" to complete may be forced to abort and start over. The nbMontage system as a whole is lock free; sync is actually wait free.

4.3 Durable Strict Serializability

Linearizability, of course, is not suitable for transactions, which must remain speculative until all operations can be made visible together. STM systems typically provide strict serializability instead: transactions in a crash-free history appear to occur in a sequential order that respects real time (if A commits before B begins, then A must serialize before B) [34, Sec. 3.1.2]. For a persistent version of NBTC, we need to accommodate crashes.

Like Izraelevitz et al. [21], we assume a full-system crash failure model: data structures continue to exist after a crash, but are accessed only by new threads—the old threads disappear. Under this model:

Definition 4.1. An execution history H displays *durable strict serializability* (DSS) if it is strictly serializable when crash events are elided.

Like durable linearizability, this definition requires all work completed before a crash to be visible after the crash. The buffered analogue is similar:

Definition 4.2. An execution history H displays *buffered durable strict serializability* (BDSS) if there exists a happens-before–consistent cut of each inter-crash interval such that H is strictly serializable when crash events are elided *along with the post-cut suffix of each inter-crash interval.*

4.4 Merging Medley with nbMontage

The epoch system of nbMontage provides a natural mechanism with which to provide failure atomicity and durability for Medley transactions: if operations of the same transaction always occur in the same epoch, then they will be recovered (or lost) together in the wake of a crash.

Building on this observation, we merge the two systems to create *txMontage*. Payloads of all operations in a given transaction are labeled with the same epoch number. That number is then validated along with the rest of the read set during Medley-style MCNS commit, ensuring that the transaction commits in the expected epoch. With nbMontage-style periodic persistence, those payloads are guaranteed to become persistent atomically.

While nbMontage itself is quite complex, this additional epoch number validation is all that is required to graft it (and all its converted, persistent data structures from the nbMontage code suite) onto Medley: persistence comes into transactions "almost for free." The transform from an nbMontage data structure into a transactional txMontage structure is the same as in Medley discussed at the beginning of Sec. 3.1.

5 CORRECTNESS

In this section, we argue that histories comprising well-formed Medley transactions are strictly serializable, that Medley is obstruction free, and that txMontage provides buffered durable strict serializability.

Definition 5.1. A Medley transaction is *well-formed* if

(1) it starts with txBegin and ends with txEnd, optionally with txAbort in between;
(2) it contains operations of Medley-transformed data structures; and
(3) all other intra-transaction "glue" code is nonblocking and free from any side effects not managed by handlers for the TransactionAborted exception.

5.1 Strict Serializability

Lemma 1. *At the implementation level (operating on the array of words that comprises system memory), nbtcLoad, nbtcCAS, tryFinalize, txAbort, and txEnd (MCNS) are linearizable operations.*

Proof (sketch). Follows directly from Harris et al. [16]. Their RDCSS compares (without changing) only a single location, and their CASN supports the update of *all* touched words, but the proofs adapt in a straightforward way. In particular, as in RDCSS, an unsuccessful tryFinalize or txEnd can linearize on a (failed) validating read or a failed CAS of its status word. A tryFinalize or txEnd whose status CAS is successful linearizes "in the past," on the first of its validating reads. (Ironically, this means that MCNS, at the implementation level, does not have an immediately identifiable linearization point.) □

Lemma 2. *In any history in which transaction t performs an nbtcLoad or nbtcCAS operation x on CASObj o, and in which t's txEnd operation y succeeds, no tryFinalize or txEnd for a different transaction that modifies o succeeds between x and y.*

Proof (sketch). Suppose the contrary, and call the transaction with the conflicting tryFinalize or txEnd u. If u's nbtcCAS of o occurs between x and y, it will abort and uninstall t's descriptor, or cause read validation to fail in y, contradicting the assumption that

t's txEnd succeeds. If u's nbtcCAS of o occurs before x, then x will abort and uninstall u's descriptor, contradicting the assumption that u's tryFinalize or txEnd succeeds after x. □

Theorem 3. *Histories comprising well-formed Medley transactions are strictly serializable.*

Proof (sketch). In an Medley-transformed data structure, all critical memory accesses will be performed using nbtcLoad or nbtcCAS. These will be followed, at some point, by a call to txEnd. If that call succeeds, no conflicting tryFinalize or txEnd succeeds in the interim, by Lemma 2. This in turn implies that our Medley history is equivalent to a sequential history in which each operation takes effect at the nbtcLoad or nbtcCAS corresponding to the linearization point of the original data structure operation, prior to NBTC transformation. Moreover, all operations of the same transaction are contiguous in this sequential history—that is, our Medley history is strictly serializable. □

5.2 Obstruction Freedom

Theorem 4. *When used to build well-formed transactions that retry on abort, Medley is obstruction free.*

Proof (sketch). In any reachable system state, if one thread continues to execute while others are paused, every nbtcLoad or nbtcCAS that encounters a conflict will first finalize (commit or abort) the encountered descriptor, uninstall it, and install its own descriptor. If the thread encounters its own descriptor, a nbtcLoad will return the speculated value and a nbtcCAS will update the write set if the argument matches the previous new value in the write set. In either case, the MCNS will make progress. If it eventually aborts, it may repeat one round of a brand new MCNS which, with no newly introduced contention, must commit. □

5.3 Buffered Durable Strict Serializability

Theorem 5. *Histories comprising well-formed txMontage transactions exhibit buffered durable strict serializability.*

Proof (sketch). Each transaction reads the current epoch, e, in txBegin. It then validates this epoch number during MCNS commit. Per Lemma 1, this MCNS must linearize inside e. With nbMontage-provided failure atomicity of all operations in the same epoch, the theorem trivially holds. □

6 PERFORMANCE RESULTS

As noted in Section 1, we have used Medley to create NBTC versions of Michael and Scott's queue [29], Fraser's skiplist [10], the rotating skiplist of Dick et al. [7], Michael's chained hash table [28], and Natarajan and Mittal's binary search tree [31]. All of the transformations were straightforward. In this section we report on the performance of Medley and txMontage hash tables and skiplists, comparing them to various alternatives from the literature. Source code for Medley, txMontage, and the experiments is available at https://github.com/urcs-sync/Medley.

Specifically, we tested the following transient systems:

Medley – as previously described (hash table and skip list)
OneFile – transient version of the lock-free STM of Ramalhete et al. [33] (hash table and skip list)
TDSL – transactional data structure library of Spiegelman et al. [36] (authors' skiplist only)
LFTT – lock-free transactional transform of Zhang et al. [43] (authors' skiplist only)

We also tested the following persistent systems:

txMontage – Medley + nbMontage (hash table and skiplist)
POneFile – persistent version of OneFile [33] (hash table and skiplist)

6.1 Experimental Setup

We report throughput for hash table and skiplist microbenchmarks and for skiplists used to run a subset of TPC-C [6]. We also measure latency for skiplists. All code will be made publicly available prior to conference publication.

All tests were conducted on a Linux 5.3.7 (Fedora 30) server with two Intel Xeon Gold 6230 processors. Each socket has 20 physical cores and 40 hyperthreads, totaling 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 on socket 1. Each socket has 6 channels of 32 GB DRAMs and 6 channels of 128 GB Optane DIMMs. We mount NVM from each socket as an independent ext4 file system. In all experiments, DRAM is allocated across the two sockets according to Linux's default policy; in persistent data structures, only NVM on socket 0 is used, in direct access (DAX) mode. In all cases, we report the average of three trials, each of which runs for 30 seconds.

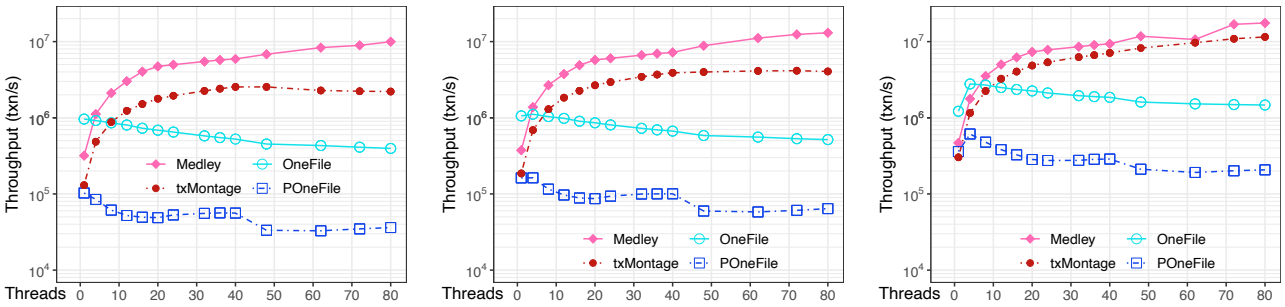
Our throughput and latency microbenchmark begins by preloading the structure with 0.5 M key-value pairs, drawn from a key space of 1 M keys. Both keys and values are 8-byte integers. In the benchmarking phase, each thread composes and executes transactions comprising 1 to 10 operations each. Operations (on uniformly random keys) are chosen among get, insert, and remove in a ratio specified as a parameter (0:1:1, 2:1:1, or 18:1:1 in our experiments).

In OneFile, we use a sequential chained hash table parallelized using STM. In Medley, we use an NBTC-transformed version of Michael's lock-free hash table [28]. Each table has 1 M buckets. In OneFile and TDSL, skiplists are derived from Fraser's STM-based skiplist [10]. In LFTT and Medley, they are derived from Fraser's CAS-based nonblocking skiplist [10]. Each skiplist has up to 20 levels.

For TPC-C, we are limited by the fact that Fraser's skiplists do not support range queries. Following the lead of Yu et al. [42] in their experiments with DBx1000[42], we limit our experiments to TPC-C's newOrder and payment transactions, which we perform in a 1:1 ratio. These are the dominant transactions in the benchmark; neither performs a range query.

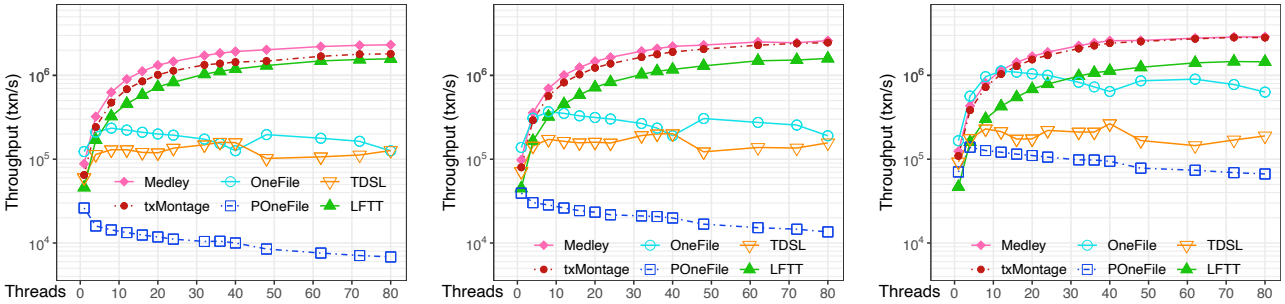
6.2 Throughput (Transient)

Throughput results for the hash table and skiplist microbenchmarks appear in Figures 7 and 8, respectively. Solid lines represent transactions on transient data structures; dotted lines represent persistent transactions. Considering only the transient case for now, Medley



(a) get:insert:remove 0:1:1 (b) get:insert:remove 2:1:1 (c) get:insert:remove 18:1:1

Figure 7: Throughput of transactional hash tables (log Y axis).



(a) get:insert:remove 0:1:1 (b) get:insert:remove 2:1:1 (c) get:insert:remove 18:1:1

Figure 8: Throughput of transactional skiplists (log Y axis).

consistently outperforms the transient version of OneFile by more than an order of magnitude, on both hash tables and skiplists, for anything more than a trivial number of threads. The gap becomes larger when the workload has a higher percentage of writes. Despite its lack of scalability, OneFile performs well at small thread counts, especially with a read-mostly workload. We attribute this fact to its serialized transaction design, which eliminates the need for read sets.

As described in Section 1, TDSL provides (blocking) transactions over various specially constructed data structures. While conflicts still occur on writes, read sets are limited to only semantically critical nodes, and the authors report significant improvements in throughput relative to general-purpose STM [36]. As shown in Figure 8, however, TDSL, like OneFile, has limited scalability, and is dramatically outperformed by Medley. Somewhat to our surprise, TDSL also fails to outperform OneFile on this microbenchmark, presumably because of the latter's elimination of read sets.

Among the various skiplist competitors, LFTT comes closest to rivaling Medley, but still trails by a factor of 1.4–2× in the best (write-only) case. Re-executing entire transactions in LFTT introduces considerable redundant work—planning in particular. On read-mostly workloads, where Medley benefits from invisible readers, LFTT trails by a factor of 2–2.7×.

As a somewhat more realistic benchmark, we repeated our comparison of Medley, OneFile, and TDSL on the newOrder and payment transactions of TPC-C. We were unable to include LFTT in these tests because it supports only static transactions, in which the set of data structure operations is known in advance—nor could we integrate its dynamic variant (DTT [23]), as the available version of the code does not allow arbitrary key and value types. LaBorde et al. [23] report, however, that DTT's performance is similar to that of LFTT on simple transactions. Given that DTT has to publish the entire transaction as a lambda expression on all its critical nodes, we would expect DTT's performance to be, if anything, somewhat worse on the large transactions of TPC-C, and LFTT was already about 2× slower than Medley on the microbenchmark.

TPC-C throughput for Medley, (transient) OneFile, and TDSL appears in Figure 9. Because transactions on TPC-C are large, OneFile is impacted severely. By ensuring the atomicity of only critical accesses, Medley still scales for large numbers of threads and outperforms the competition by as much as 45×.

6.3 Latency (Transient)

In an attempt to assess the marginal cost of transaction composition, we re-ran our microbenchmark on Fraser's original skiplist (**Original**—no transactions), the NBTC-transformed skiplist without transactions (**TxOff**—no calls to txBegin or txEnd), and the NBTC-transformed skiplist with transactions (**TxOn**—as in Figure 8).

Figure 10a reports latency for structures placed in DRAM. Without transactions, the transformed skiplist is 1.8× slower than the original. With transactions turned on, it's about 2.2× slower. These

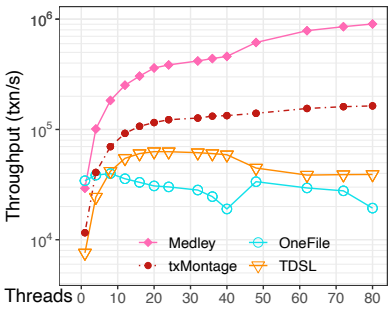


Figure 9: TPC-C skiplist throughput (log Y axis).

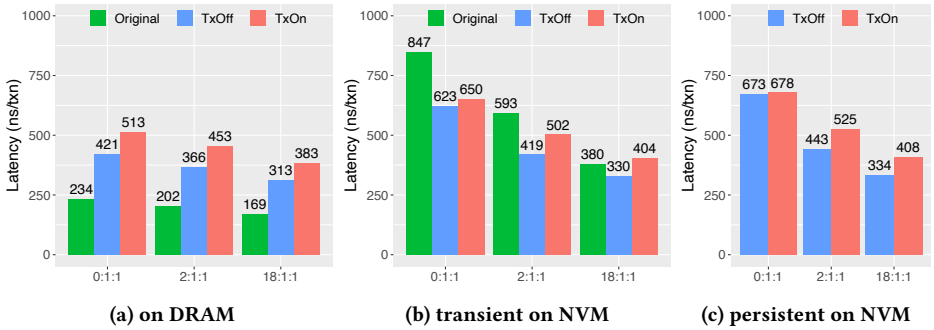


Figure 10: Average latency on skiplists at 40 threads. X labels are ratio of get:insert:remove.

results suggest that the more-than-doubled cost of CASes (installing and uninstalling descriptors) accounts for about $^2/_3$ of Medley's overhead.

6.4 Persistence

To evaluate the impact of failure atomicity and durability on the throughput of txMontage, we can return to the dotted lines of Figures 7, 8, and 9.

Throughput. In the microbenchmark tests, with strict persistence and eager cache-line write-back, persistent OneFile is an order of magnitude slower than its transient version. With periodic persistence, however, the txMontage hash table achieves half the throughput of Medley at 40 threads on the write-only workload—almost two orders of magnitude faster than POneFile. With a read-mostly workload on the hash table, or with any of the workloads on the skiplist (with its lower overall concurrency), txMontage is almost as fast as Medley. In the extreme write-heavy case (80 threads on the 0:1:1 hash table workload), we attribute the roughly 4× slowdown of txMontage to NVM's write bottleneck [22]—in particular, to the phenomenon of *write amplification* [20, 41].

Results are similar in TPC-C (Fig. 9). Transactions here are both large and heavy on writes; allocating payloads on NVM limits txMontage's throughput to roughly a fifth of Medley's, but that is still about 4× faster than transient OneFile. POneFile, for its part, spent so long on the warm-up phase of TPC-C that we lost patience and killed the test.

Latency. Figure 10b shows the latency of skiplist transactions when txMontage payloads are allocated on NVM (and indices on DRAM) but persistence is turned off (no epochs or explicit cache line write-back). For comparison, we have also shown the latency of the original, non-transactional skiplist with *all* data placed in NVM. Figure 10c shows the corresponding latencies for fully operational txMontage.

Comparing Figures 10a and 10b, we see lower marginal overhead for transactions when running on NVM. This may suggest that the hardware write bottleneck is reducing overall throughput and thus contention.

On the write-only workload (leftmost groups of bars), moving payloads to NVM introduces an overhead of almost 50% (Fig. 10a versus Fig. 10b). On the read-mostly workload (rightmost bars), this penalty drops to 5%. Again, we attribute the effect to NVM's write bottleneck. The high latency of the original skiplist entirely allocated on NVM (green bars in Figure 10b) appears to confirm this hypothesis.

Comparing Figures 10b and 10c, txMontage pays less than 5%, relative to Medley on NVM, for failure atomicity and durability.

7 CONCLUSION

We have presented nonblocking transaction composition (NBTC), a new methodology that leverages the linearizability of existing nonblocking data structures when building dynamic transactions. As concrete realizations, we introduced the *Medley* system for transient structures and the *txMontage* system for (buffered) persistent structures. Medley transactions are isolated and consistent; txMontage transactions are also failure atomic and durable. Both systems are quite fast: where even the best STM has traditionally suffered slowdowns of 3–10×, Medley incurs more like 2.2×; txMontage, for its part, adds only 5–20% to the overhead of nbMontage, allowing it to outperform existing nonblocking persistent STM systems by nearly two orders of magnitude.

Given their eager contention management, Medley and txMontage maintain obstruction freedom for transactions on nonblocking structures. In future work, we plan to explore lazy contention management, postponing installment of descriptors until transactions are ready to commit. By sorting and installing descriptors in canonical order, the resulting systems would preserve lock freedom. Lazy contention management would also facilitate helping, as any installed descriptor would have `status == InProg`, and any other thread could push it to completion.

As currently defined in NBTC, *speculation intervals* are easy to identify, but may unnecessarily instrument certain harmless helping instructions between publication and linearization. We are currently working to develop a more precise but still tractable definition of helping in order to reduce the number of "critical" memory accesses that must be performed atomically in each transaction.

REFERENCES

[1] 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)*. virtual conference, 283–293.
[2] Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du, Rafaello Sanna, Shreif Abdallah, and Michael L. Scott. 2021. Fast Nonblocking Persistence for

Concurrent Data Structures. In *35th Intl. Symp. on Distributed Computing (DISC)*. Freiburg, Germany, 14:1–14:20.

[3] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *Usenix Annual Technical Conf. (ATC)*. virtual conference, 799–812.

[4] 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)*. Newport Beach, CA, 105–118.

[5] 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)*. Vienna, Austria, 271–282.

[6] The Transaction Processing Council. 2010. TPC-C Benchmark (Revision 5.11.0). http://www.tpc.org/tpcc/.

[7] Ian Dick, Alan Fekete, and Vincent Gramoli. 2016. A Skip List for Multicore. *Concurrency and Computation: Practice and Experience* 29, 4 (May 2016), 20 pages.

[8] Avner Elizarov, Guy Golan-Gueta, and Erez Petrank. 2019. LOFT: Lock-Free Transactional Data Structures. In *24th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Washington, DC, 425–426. Extended version available as the Technion M.Sc. technical report MSC-2019-01, November 2018. www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/2019/MSC/MSC-2019-01.

[9] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *31st ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. Phoenix, AZ, 275–286.

[10] Keir Fraser. 2003. *Practical Lock-Freedom.* Ph. D. Dissertation. King's College, Univ. of Cambridge. Published as Univ. of Cambridge Computer Laboratory technical report #579, February 2004. www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[11] 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 Conf. on Programming Language Design and Implementation (PLDI)*. virtual conference, 377–392.

[12] 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)*. Vienna, Austria, 28–40.

[13] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: Making Lock-Free Data Structures Persistent. In *42nd ACM Conf. on Programming Language Design and Implementation (PLDI)*. virtual conference, 1218–1232.

[14] Rachid Guerraoui and Michal Kapalka. 2008. On the Correctness of Transactional Memory. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Salt Lake City, UT, 175–184.

[15] Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-Oriented Database Recovery. *Comput. Surveys* 15, 4 (Dec. 1983), 287–317.

[16] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *16th Intl. Symp. on Distributed Computing (DISC)*. Toulouse, France, 265–279.

[17] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of Memory Reclamation for Lockless Synchronization. *Journal of Parallel Distributed Computing (JPDC)* 67, 12 (Dec. 2007), 1270–1285.

[18] Maurice Herlihy and Eric Koskinen. 2008. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Salt Lake City, UT, 207–216.

[19] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *22nd ACM Symp. on Principles of Distributed Computing (PODC)*. Boston, MA, 92–101.

[20] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A Hybrid PMem-DRAM Persistent Hash Index with Fast Recovery. In *Intl Conf on Management of Data (SIGMOD)*. Philadelphia, PA, 1049–1063.

[21] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *30th Intl. Symp. on Distributed Computing (DISC)*. Paris, France, 313–327.

[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 preprint arXiv:1903.05714v3.

[23] Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. 2019. Wait-Free Dynamic Transactions for Linked Data Structures. In *10th Intl. Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. Washington, DC, 41–50.

[24] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China, 329–343.

[25] Virendra Jayant Marathe and Mark Moir. 2008. Toward High Performance Nonblocking Software Transactional Memory. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Salt Lake City, UT, 227–236.

[26] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. 2006. Lowering the Overhead of Software Transactional Memory. In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*. Ottawa, ON, Canada, 11 pages.

[27] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. 2001. Read Copy Update. In *Ottawa Linux Symp.* Ottawa, ON, Canada, 338–367.

[28] Maged M. Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *14th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. Winnipeg, MB, Canada, 73–82.

[29] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *15th ACM Symp. on Principles of Distributed Computing (PODC)*. Philadelphia, PA, 267–275.

[30] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*. Seattle, WA, 35–46.

[31] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Orlando, FL, 317–328.

[32] 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 *31st Intl. Symp. on Distributed Computing (DISC)*. Vienna, Austria, 37:1–37:16.

[33] 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)*. Portland, OR, 151–163.

[34] Michael L. Scott. 2013. *Shared-Memory Synchronization.* Morgan & Claypool Publishers, San Rafael, CA.

[35] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. 2009. A Comprehensive Contention Management Strategy for Software Transactional Memory. In *14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Raleigh, NC, 141–150.

[36] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *37th ACM Conf. on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA, 682–696.

[37] Fuad Tabba, Mark Moir, James R. Goodman, Andrew W. Hay, and Cong Wang. 2009. NZTM: Nonblocking Zero-indirection Transactional Memory. In *21st ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. Calgary, AB, Canada, 204–213.

[38] Shahar Timnat and Erez Petrank. 2014. A Practical Wait-Free Simulation for Lock-Free Data Structures. In *19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. Orlando, FL, 357–368.

[39] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *24th ACM Symp. on Operating Systems Principles (SOSP)*. Farminton, PA, 18–32.

[40] 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)*. Newport Beach, CA, 91–104.

[41] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *53rd Intl. Symp. on Microarchitecture (MICRO)*. virtual conference, 496–508.

[42] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. of the VLDB Endowment* 8, 3 (Nov. 2014), 209–220.

[43] Deli Zhang and Damian Dechev. 2016. Lock-Free Transactions without Rollbacks for Linked Data Structures. In *28th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. Pacific Grove, CA, 325–336.

[44] 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), 128:1–128:26.