Type-Aware Transactions for Faster Concurrent Code

Nathaniel Herman
Harvard University/Dropbox
nherman@post.harvard.edu

Jeevana Priya Inala
MIT
jinala@mit.edu

Yihe Huang Lillian Tsai
Harvard University
yihehuang@g.harvard.edu
lilliantsai@college.harvard.edu

Eddie Kohler
Harvard University
kohler@seas.harvard.edu

Barbara Liskov
MIT
liskov@piano.csail.mit.edu

Liuba Shrira
Brandeis University
liuba@brandeis.edu

Abstract

It is often possible to improve a concurrent system's performance by leveraging the semantics of its datatypes. We build a new software transactional memory (STM) around this observation. A conventional STM tracks read- and write-sets of memory words; even simple operations can generate large sets. Our STM, which we call STO, tracks abstract operations on transactional datatypes instead. Parts of the transactional commit protocol are delegated to these datatypes' implementations, which can use datatype semantics, and new commit protocol features, to reduce bookkeeping, limit false conflicts, and implement efficient concurrency control. We test these ideas on the STAMP benchmark suite for STM applications and on our own prior work, the Silo high-performance in-memory database, observing large performance improvements in both systems.

1. Introduction

Transactions simplify concurrent programming by limiting interactions among threads. Transactional memory (TM) [26] brings this power and ease of use to general-purpose multicore parallel programming. TM transactions are linearizable [30]; they run as if isolated and atomic, and aborted transactions have no effect on global state. A TM programmer (or TM compiler) labels accesses to shared memory and surrounds them with "transaction" blocks. The TM system takes care of everything else, from concurrency control to deadlock prevention.

EuroSys '16 April 18–21, 2016, London, United Kingdom
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901348

However, TMs have performance issues. In hardware TM, fundamental microarchitectural limitations, such as bounds on maximum transaction size, mean some valid transactions can never commit [51]. Implementations will gradually improve, but general-purpose transactions must be backed up by software. Unfortunately, software TM performance severely lags that of purpose-built concurrent code [7], since STM implementations have high costs for bookkeeping and concurrency control. Transaction systems must track all objects accessed during transaction execution, either by locking them or by taking snapshots for later validation, and in STM, these objects are typically memory words. This is universal, since every concurrent data structure is stored in memory, but expensive; for instance, a binary search tree lookup must track every word accessed in the path from the root. Word-based bookkeeping has high overhead (the tracking set contains many words that must be tracked and validated). It also makes concurrency control more expensive by making conflicts more likely (any concurrent change to the path will cause the transaction to abort, including rotations with no semantic effect on the lookup).

Our work makes STM faster and more general by basing it instead on abstract datatype operations. We call our design *STO*, for *software transactional objects*. STO's commit protocol works with abstract reads, writes, and predicates on transactional datatypes, rather than concrete accesses to untyped memory. Datatype callbacks perform all concrete locking, version verification, and data structure modification. This separation of concerns lets STO track hundreds of times fewer objects than TL2 [12], a popular STM implementation.

STO supports datatypes from vectors and trees to hash tables and priority queues. Datatypes can leverage their semantics to reduce false conflicts and improve scalability. For instance, in a conventional STM, all increments to a transactional counter conflict (since increment both reads and writes the counter). In STO, in contrast, a counter implementer can take advantage of increment's commutative semantics: though increments will serialize at commit time, concurrent increments need not abort. New features of our commit protocol, such as optimistic predicate verification, give datatypes further power to avoid conflicts.

STO comprises a core system that implements the commit protocol and an extensible library of transactional datatypes. Most programmers will not need to implement new datatypes, since they can use those available in libraries; we hope this makes it easier to write transactional programs that both avoid bugs and perform well. However, advanced programmers can add new datatypes, either general or application-specific [32]. As with any efficient concurrent datatype, a STO transactional datatype can be hard to write, but STO's helper classes and design patterns ease this burden and our experience argues this skill can be learned.

Other STM systems have partially integrated concurrent datatypes, some using new layers of concurrency control, such as pessimistic lock tables and undo logs [25] or predicate tables of STM words [2], and others co-developing a commit protocol and a datatype implementation [23, 24]. STO has lower overhead than many of these systems, but it also can support their ideas separately or in combination. For instance, some STO datatypes gain performance by mixing aspects of optimistic and pessimistic concurrency control.

STO scales and performs well. On a variety of STAMP benchmarks, it exceeds the performance of the fast TL2 software transactional memory by factors ranging from 1.4x to 118x. The largest speedups were achieved using STO's support for application-specific conflict reduction. We also match or outperform transactional boosting [25]. Finally, our reimplementation of the very fast Silo in-memory database [48] exceeds Silo's performance by 1.23x or more on the TPC-C benchmark, even though Silo was heavily optimized. Our reimplementation contains far less code, and that code is easier to understand.

Our contributions are:

- The STO system: a lean, fast transactional memory system for C++ that supports arbitrary transactional datatypes (§3).
- A library of easy-to-use transactional datatypes for STO, including lists, hash tables, and trees (§4).
- Design patterns for co-designing concurrent data structures with a transactional memory.
- Our evaluation and our STO-Silo implementation (§5).

2. Related work

Transactional memory [26] aims to simplify concurrent programming. Though the performance of transactional memory systems has been extensively optimized [22], overheads due to bookkeeping and false conflicts can lead to unsatisfactory performance [6, 7], even in high-performance TMs like TL2 [12] and LarkTM [52].

The SwissTM transactional memory [14] improves on TL2's performance using several techniques. It tracks memory in 4-word groups, which has up to 4x less bookkeeping overhead than single-word tracking but fewer false conflicts than coarser cache-line tracking. STO uses datatype integration to achieve a similar goal, but datatype semantics let us reduce bookkeeping further (by up to 40x) without inducing false conflicts. SwissTM's "mixed invalidation" scheme has some of the advantages of both pessimistic and optimistic concurrency control. Write/write conflicts are detected early (written locations are pessimistically locked during transaction execution), while read/write conflicts are detected lazily (concurrent reads can often proceed despite locks). Mixed invalidation is complementary to our work; we plan to evaluate it in our datatypes. Although mixed invalidation can reduce false conflicts, SwissTM, like any word-based STM, cannot support STO's more advanced forms of application-specific conflict prevention.

Non-transactional APIs, such as open nesting [40], elastic transactions [16], transactional collection classes [5], and early release [28], can eliminate some false conflicts, and other proposals, such as SpecTM [13], allow programmers to reduce the bookkeeping costs for small subsets of transactions, such as those that read and write four or fewer memory locations. Both these lines of work complicate the TM abstraction.

Object-based STMs [19, 29] work at the object level rather than the word level. However, unlike STO, these systems make shadow copies of any modified objects. These copying costs can be large, and false conflicts are not reduced. Multi-version protocols such as JVSTM [3, 17, 18] can reduce conflicts for read-only transactions, but maintain even more copies.

Our work most relates to prior research that speeds up memory transactions using abstract data types, including boosting [25], automatic locking [20], and predication [2]. These techniques are based on concurrent data structures written outside the STM. Relying on datatypes for concurrency control lets these systems exploit operation commutativity, avoid false conflicts, and reduce conventional STMs' single-thread overheads, but to implement transactional semantics, the systems duplicate some work their datatypes already perform.

Boosting [25] wraps unmodified concurrent datatype implementations for use in STM. Transactional semantics is ensured by a separate table of so-called *abstract locks*. Boosting wrapper functions map each operation onto the corresponding abstract locks, which are acquired before the underlying operation is called and held until the transaction commits or aborts. Thus, the transactional wrapper for a map insert operation would acquire the abstract lock corresponding to the requested key. If operations access distinct abstract locks, they can take place concurrently, so datatype semantics determine the abstract lock mapping. Wrappers use undo logging to ensure modifications are undone if the transaction aborts, so every wrapped operation must have an inverse. Automatic locking [20] also synthesizes a boosting-style abstract lock mapping from commutativity specifications; a similar approach could potentially help STO datatype authors choose transaction items (see §3.2).

Predication [2] also supports transactional access to fast concurrent datatypes via a shared-memory concurrency control table, but this table consists of memory words controlled by the underlying conventional STM. Wrapper operations access STM "predicate words," where each predicate word corresponds to a semantic property of the set or map object (such as the absence of a key). A lookup on an absent key inserts a predicate word into this table, introducing a difficult garbage collection problem STO avoids.

These systems complement STO, whose core commit protocol is flexible enough to support their techniques individually and in combination. Although a boosting implementation strategy can be somewhat simpler than full transactional integration, STO's approach offers advantages; unlike boosting, for example, STO datatypes can easily support modifications that have no inverse.

STO, like all transaction designs using abstract datatypes, builds on foundational work by Weihl [49], who described transactional atomicity in terms of datatype semantics. Weihl defines a *local atomicity* property implementations must satisfy to guarantee transaction serializability, while taking commutativity into account. He considers some aspects of how operation semantics affect implementation requirements, such as history log compaction or view computation, but does not focus on concrete implementation.

Our methods for avoiding false conflicts have been used in other contexts. In addition to Weihl, early work on reducing transaction conflicts by exploiting commutativity and other semantic properties include Schwartz and Spector [45], Badrinath and Ramamritham [1], and Korth [31]. More recent techniques on exploiting commutativity in replicated data types include work on CRDTs [46] and much other research in the database community. This work is complementary with ours: a STO datatype designer can use commutativity reasoning to improve the performance of in-memory transactions.

The Galois system [32, 33] proposes a framework in which individual data structures specify operations that can and cannot commute, and uses it to improve the performance of two irregular parallel graphics applications. This demonstrates that type-specific concurrency control of the kind supported by STO is rich with possibilities beyond fundamental datatypes. Galois focused not on composing serializable transactions, but on a systematic method for constructing commutativity checkers for type-specific conflict detection; this work is complementary to ours.

STO's flexibility lets datatype designers adopt many ideas from the literature on concurrent datatype design, including locking, optimistic verification, direct updates (eager versioning), and deferred updates (lazy versioning) [22, 27]. Our datatypes take advantage of this flexibility; for instance, our map data structures use direct updates for inserted items and deferred updates for all other modifications (§4), and our list datatypes use some techniques like those in the optimistic transactional lazy set [24]. Lock-free compare-and-swap designs tend to interact badly with transactions, however, since transactional commit protocols use locking to ensure that multiple modifications appear to commit atomically.

Recent work on reducing bookkeeping and conflict avoidance costs in STM includes techniques for exploiting hardware transactional memory [4, 10, 11, 37, 50]. These efforts focus on structuring transactions to reduce conflicts and minimize the transaction data footprint so they can run in hardware. This work is orthogonal to the issues discussed here, and we plan to investigate combining STO with HTM. For instance, a datatype could use HTM to atomically read a value and its associated version number.

3. Design

This section describes the design and implementation of STO's transaction system.

3.1 Overview

STO is a C++ library. STO user transactions are C++ code blocks, delimited by `TRANSACTION` and `RETRY`, that invoke operations on *transactional objects*. Transactions have linearizable effects: operations invoked by uncommitted transactions aren't visible elsewhere, and operations invoked by committed transactions have the same effects as if the committed transactions ran in some serial order. Aside from the `TRANSACTION` blocks, STO programming feels like conventional C++ programming. Transactional datatypes are compatible with the C++ standard library and support programming patterns such as iterators.

Here is a STO balance-transfer function:

```
bool transfer(TBox<int>& bal1, TBox<int>& bal2,
              int amt) {
    TRANSACTION {                     // open new transaction
        if (amt < 0 || bal1 < amt)
            return false;
        bal1 = bal1 - amt;
        bal2 = bal2 + amt;
    } RETRY(true);                       // commit with retry
    return true;
}
```

The `bal1` and `bal2` variables are transactional objects with transactional datatype `TBox<int>`. This type is a transactional proxy for an `int`; its contents are the `int` and a version number/lock field that supports safe concurrent access and transactional bookkeeping (§3.4 has details). `TBox` provides transactional assignment and convert-to-integer operations, so the expression `bal1 = bal1 - amt` contains a transactional read of the current balance and a transactional write of the new balance. STO provides opacity (§3.6), so the transaction will never observe inconsistent state.

Transactions can also access non-transactional objects such as `amt`. To maintain transactional correctness, these objects should generally be thread-local or immutable.

STO's commit protocol runs at the end of a `TRANSACTION` block. On success, any writes are installed into the transactional objects, making them visible to other transactions. On failure (for instance, if a concurrently-executing transaction causes a conflict), the writes are discarded and STO evaluates the `RETRY` expression. If the expression is true, STO retries the transaction; if false, STO throws an `Abort` exception. This retry check also occurs when the transaction aborts before the commit protocol begins, either because the user initiates an abort or because the transaction observes inconsistent state (see §3.6). If control leaves the block in another way (by `return`, `break`, or an exception other than `Abort`), STO causes the transaction to abort with no retry. A hidden guard object ensures such aborts are clean.

3.2 STO platform

STO comprises a shared core and an extensible library of transactional datatypes. Figure 1 introduces the core interface, and Figure 2 shows this interface in use in the `TBox` datatype's implementation.

The core implements the commit protocol and provides common functionality used by datatypes, such as operations on version numbers. Its central feature is an efficient per-transaction *tracking set* in which datatypes record information for the commit protocol.

Transactional datatypes are concurrent datatypes that provide transactional operations. All transactional datatypes inherit from `TObject`, which defines a callback interface for the commit protocol, but user code never calls `TObject` methods directly. STO's library of transactional datatypes includes `TBox<T>`, counters, arrays, hash tables, linked lists, binary search trees, Masstree [36], and an implementation of word-based STM; advanced users can add their own.

A transactional datatype is, first and foremost, a concurrent datatype—it must support safe concurrent access from multiple threads—but its operations must additionally be implemented in a transaction-safe way. This imposes three main requirements: version numbers, tracking set integration, and commit callbacks.

First, transactional datatypes must use *version numbers* to track updates. A version number comprises an ID and some helper bits, including a lock bit. The ID is the ID of the committed transaction that most recently changed the corresponding state. When the lock bit is set, the corresponding state is being updated and is unsafe to access. Many datatypes reduce conflicts by assigning multiple version numbers per object. Each version number typically protects a different logical segment of state, where segments are defined so that operations accessing disjoint sets of segments commute. For instance, a transactional array has one logical segment, and one version number, per index.

Second, datatypes' transactional operations must integrate with the STO core's *tracking set*. Assuming optimistic validation with delayed updates, this integration requires three things. (1) When a transactional operation observes object state, it must record information about what was read (typically a version number) in the tracking set. (2) When a transactional operation would logically modify object state, it must record information about the modification in the tracking set. (3) When a transactional operation observes state that it may have modified itself, it must check the tracking set for an outstanding modification and return the corresponding result. STO supports other concurrency control designs, but the principle is the same: any commit- or abort-time actions, such as validation, installation, and rollback, are recorded in the tracking set.

The tracking set's representation is an array of *transaction items* of class `TItem`. Tracking sets are thread-local—each transaction has its own, and each thread runs at most one transaction at a time—so `TItems` need not protect against concurrent access. `Sto::item(owner, key)` returns the unique `TItem` in the current thread's tracking set identified by `owner` and `key`, where `owner` is a pointer to a transactional object and `key` is an object of arbitrary type (but usually an integer or pointer). Keys distinguish between logical segments of the `owner` object's state; for instance, our array's items use element indexes as keys. If the requested `TItem` doesn't exist yet, the `Sto::item` method initializes it and adds it to the current transaction's tracking set.

A `TItem` can contain a *read version*, a *write value*, and a *predicate value*. The presence of a read version or predicate value tells STO that the item should be verified during the commit protocol, and the presence of a write value indicates that the item should be locked during the commit protocol and the `install` method should be called if the transaction commits. `TItems` also offer several bits of flags available for datatype use. `TItems` are initially empty: there are no versions or values and all flags are zero. Read version and predicate value are mutually exclusive.

`TItem` predicate values, write values, and keys have arbitrary type. Small objects, such as pointers and integers, are stored directly in the `TItem`. Larger objects are stored in a separate per-transaction memory buffer, with the item holding pointers into the buffer; STO ensures the buffered values are properly destroyed when the transaction completes. This lets STO support arbitrary values without common-case overhead. It also turns `TItem` into a union that can hold objects of any type. Datatype implementations must ensure that they extract objects from `TItems` using the right types; for instance, to access the key of an item created with `Sto::item(o, 1)`, a datatype should call `item.key<int>()`.

Finally, transactional datatypes must define *callbacks* for the commit protocol. STO commits transactions using a pro-

```
namespace Sto:
  // look up a transactional item, creating if necessary:
  TItem item<K>(TObject* owner, K key);
  // return the committing transaction's version:
  TVersion commit_id() const;

class TItem:              // transactional item; size 4 words (32B)
  TObject* owner() const;
  K key<K>() const;

  bool has_read() const;                          // read tracking
  // observe a version number and check opacity:
  void observe(TVersion vers);
  // atomically read value and observe vers:
  R read<R>(TWrapped<R>& value, TVersion& vers);
  // check whether a version number changed:
  bool check_version(TVersion& vers) const;

  bool has_write() const;                         // write tracking
  W write_value<W>() const;
  void add_write<W>(W write_value);

  bool has_predicate() const;                 // predicate tracking
  P predicate_value<P>() const;
  void add_predicate<P>(P predicate_value);

  unsigned flags() const;                             // user flags
  void set_flags(unsigned flags);

class TObject:       // datatype superclass for commit callbacks
  // called in phase 1 to lock write items:
  bool lock(TItem it);
  // called in phase 1 to upgrade predicates:
  bool check_predicate(TItem it);
  // called in phase 2 to validate reads:
  bool check(TItem it);
  // called in phase 3 to install write items:
  void install(TItem it);
  // called after commit/abort to unlock write items:
  void unlock(TItem it);
  // called after commit/abort to clean up as necessary:
  void cleanup(TItem it, bool committed);
```

Figure 1. The core STO interface.

tocol based on optimistic concurrency control [34]. Its core functions, however, are delegated to datatype callbacks on the transaction's `TItems`. This delegation offers datatype designers considerable flexibility. For example, datatypes can use pessimistic locking for some or all updates, in which case their `lock` methods do nothing and their `cleanup` methods roll back uncommitted modifications. Datatypes are free to reorganize their memory layouts, since the STO core does not assume that locks and version numbers stay in the same place.

3.3 Basic commit protocol

We now describe STO's basic commit protocol. Some advanced features, such as predicates, are described later.

On entering a `TRANSACTION` block, STO starts a new transaction and initializes the calling thread's tracking set to empty. The commit protocol runs at the end of the `TRANSACTION` block, when the transaction tries to commit.

The protocol proceeds in phases. In **Phase 1**, all modified `TItems` are locked. This blocks other conflicting modifications until the current transaction completes, and ensures that other transactions can see that modifications are in progress. The STO core calls `it.owner()->lock(it)` for each modified item `it` (that is, each item where `it.has_write()` is true). The lock callback must lock the relevant segment of state, using bounded spinning to prevent deadlock; if a lock attempt fails, the transaction aborts.

In **Phase 2**, all read version numbers are verified to ensure that no conflicting updates have occurred. STO calls `check` for each read item `it` (that is, each item with `it.has_read()` true). The `check` callback must compare the tracking set's stored version number with the live version number in the corresponding shared state, returning true if and only if the version number hasn't changed and is not locked by any other transaction. If a version check fails, the transaction aborts.

If Phase 2 completes successfully, the transaction will commit at that point. The STO core assigns the transaction an ID by advancing a global *version clock* with an atomic increment instruction. This produces a version number guaranteed to be larger than that of any prior transaction.

In **Phase 3**, modifications are installed. STO calls `install` for each modified item `it`. This callback makes the modified value visible to other threads and updates corresponding version numbers to the transaction's ID.

The final **cleanup** phase runs whether the transaction commits or aborts. All items that were locked are unlocked via calls to `unlock` callbacks. Then STO calls the `cleanup(it, ok)` callback for every modified item, where `ok` is true if the transaction committed and false if it aborted. This callback can undo effects of aborted transactions (for datatypes that use direct updates, §4) and clean up after committed transactions (for instance, free memory that was unlinked from the data structure). They can also perform cleanup actions that aren't visible via the specification, such as data structure rebalancing. Delaying these actions to the cleanup phase is useful for scalability since it reduces the time spent holding transaction locks.

3.4 Example

Figure 2 shows how this comes together. `TBox<T>`'s transactional operations, `operator T` for transactional read and `operator=` for transactional assignment, record version numbers and write values in the box's `TItem`. (Since `TBox` has a single logical segment of state, only one `TItem` is required.) Transactional operations are chosen to ease user programming: since conversion is used to read and assignment is used to write, a `TBox<T>` can be used anywhere a `T` is expected. We also see the use of several helper classes pro-

```
class TBox<T> : public TObject {
public: // transactional operations:
   operator T() const {
      auto item = Sto::item(this, 0);
      if (item.has_write())
         return item.write_value<T>();
      else
         return item.read(value_, vers_);
   }
   TBox<T>& operator=(const T& new_value) {
      Sto::item(this, 0).write(new_value);
      return *this;
   }
   // commit callbacks:
   bool lock(TItem item) {
      return vers_.try_lock();
   }
   bool check(TItem item) {
      return item.check_version(vers_);
   }
   void install(TItem item) {
      value_.store(item.write_value<T>());
      vers_.set_version(Sto::commit_id());
   }
   void unlock(TItem item) {
      vers_.unlock();
   }
private:
   TWrapped<T> value_;
   TVersion vers_;
};
```

Figure 2. A simple transactional datatype implementation. `TBox<T>` provides transaction-safe concurrent access to an object of type `T`.

vided by the STO core. `TWrapped<T>` provides atomic access to an underlying object and `TVersion` is STO's version number class. The `item.read(value_, version_)` call atomically reads a wrapped value and version number, records the version number in `item`, and returns the value.

3.5 False conflicts and optimistic predicates

False conflicts arise when operations that don't conflict in semantic terms—that is, operations that commute—conflict in concrete terms on a shared version number. False conflicts cause unnecessary aborts, waiting, and wasted work, so for good performance, STO datatypes should avoid them.

To keep our discussion concrete, we consider a transactional counter with `increment`, `decrement`, and `test` operations, where `increment` and `decrement` return nothing and `test` returns whether the counter's value is greater than zero. In a naive implementation, all operations read the prior value of the counter, so all operations conflict.

An implementation should avoid creating conflict dependencies that aren't required by the datatype specification. As mentioned, many datatypes divide into logical segments of state, and using a version number per segment naturally reduces dependencies. But dependencies can sometimes be reduced even in datatypes with only one logical segment. For example, since our counter's `increment` and `decrement` operations return nothing, they don't expose the prior state of the counter. There is no semantic need for these operations' implementations to observe that prior state. Instead, they can record an accumulated counter delta that is applied at commit time. Update transactions still serialize on the counter's lock, but since they no longer observe counter state, concurrent updates no longer cause aborts.

Multiple version numbers can help even in datatypes that lack an obvious segment division. For instance, `test` commutes with any sequence of updates that don't change whether the counter is positive. If the counter infrequently crosses zero, a version number for zero crossings can reduce false conflicts. `Test` reads this version number; `increment` and `decrement` update the main version number at each commit, but update the zero-crossing version number only when the counter's value crosses zero. Unfortunately, this approach doesn't handle transactions that update the counter before testing it; for such transactions, `test` must fall back to the main version number.

STO's most advanced mechanism for reducing false conflicts is a new feature we call *optimistic transactional predicates*. Somewhat like predicate locks in databases [15], STO's predicates are a general methodology for avoiding false conflicts and detecting true ones.

To use a predicate, a datatype records in a `TItem` a *predicate expression* indicating conditions under which committing the transaction is acceptable. This expression is stored in a datatype-specific way. At commit time, STO calls the datatype's `check_predicate` callback to verify that the commit condition still holds; the transaction aborts if this callback fails.

In our counter, `test` can track a predicate range $[l,h]$, where the counter's `check_predicate` succeeds if and only if the commit-time counter value lies between l and h. When called on a positive counter, `test` will record the predicate $[1,\infty]$. But predicates also handle more complex sequences of operations. In a transaction in which `test` returns `false`, `increment` is called 10 times, and finally `test` returns `true`, the predicate $[-9,0]$ records a necessary and sufficient commit condition.

In optimistic commit protocols like STO's, the verification phase (Phase 2) must be able to detect all concurrent updates, including locks and update sequences that modify object state before returning it to an earlier value. This means that Phase 2 verification must consider version numbers; evaluating arbitrary predicate expressions there would not ensure atomicity in multi-operation transactions. Therefore, STO *upgrades* its predicates to version numbers during Phase 1, reducing predicate checking to conventional OCC read validation. Although false conflicts can still occur on the upgraded version numbers, they are limited in scope to the commit protocol: concurrent updates during transaction execution are ignored.

To implement the upgrade process, STO calls each predicate `TItem`'s `check_predicate` callback during Phase 1 of the commit protocol. This callback first performs an atomic read of the relevant object state and any version number(s) that cover that state. (A set of version numbers "cover" state if any change in the state is accompanied by a change in at least one of the version numbers.) It then checks the predicate using this state, returning false and aborting the transaction if the predicate no longer holds. Finally, the callback adds the covering version number(s) to the `TItem` and returns true. Once all predicates are verified, Phase 2 begins, and the upgraded version numbers are validated.

Predicates have higher overhead than version numbers due to the extra computation they add to the commit protocol. Our datatypes mostly use extra version numbers to avoid false conflicts, since this is slightly faster in low-conflict situations; our red-black tree, for example, has two version numbers per node, one for updates to data and one for updates to tree structure. Nevertheless, predicates are powerful, general, and can have high performance impact, as we'll see. Predicates also broaden the range of commutative operations STO can support.

3.6 Opacity

A transactional memory has the *opacity* property when user code never observes a transactionally inconsistent state [21, 35]. These states are dangerous because they can cause irrecoverable failures in user code, such as invalid pointer dereferences, infinite loops, and hardware traps like divide-by-zero. TM is easier to use when users need not consider such states. Opacity eliminates these states by aborting transactions as soon as they observe an inconsistency that would cause a later commit attempt to fail.

STO provides opacity using the TL2 algorithm [12]. TL2-style opacity relies on a version clock: committed transactions' version numbers must increase monotonically. STO uses the original "GV4" version clock for simplicity, but any version clock would work (and many would scale better). On starting a new transaction, STO reads the global version clock and records this as the transaction's local *version bound*. On observing transactional state, STO compares the corresponding version number with the bound. If the bound is smaller, the read is unsafe: the corresponding transaction executed in parallel with the current transaction, and thus might have invalidated previously-read state.

In TL2, unsafe version numbers always cause the transaction to abort. This can cause false aborts, however, and we observed better overall performance using revalidation [9, 42]. On observing an unsafe version number, STO reloads the transaction's version bound, then verifies all previously-read version numbers and predicates by calling the corresponding `check` and `check_predicate` methods; the transaction aborts only if one of these checks fails. (When called during an opacity check, it's safe for `check_predicate` to merely check a predicate, rather than upgrade it to a version number.) Revalidation improved our performance by an average of 14% in high-contention STAMP benchmarks, with no adverse effect at low contention.

Opacity checks must be performed on every observation of shared state. STO simplifies this task with methods like `TItem::read`, which combines an atomic read of shared state with opacity checking and tracking set management.

As a result of opacity, STO, like TL2, can commit read-only transactions without running the commit protocol. When an opaque transaction reaches the commit point, all observations saw a consistent state, so for transactions that only contained observations, committing is definitely safe. Unlike TL2, however, STO always records reads in its tracking set. This allows revalidation and isn't a large burden since STO read sets are relatively small.

As we'll see, some useful datatype implementation strategies modify shared structures before commit. This interacts poorly with global-version-clock opacity, since some version numbers are updated before the commit-time transaction ID is set. Datatype implementations can either allocate separate version clock values for such updates, or they can mark the updated version numbers as "unordered" using a reserved bit. On encountering an unordered version number, STO performs a full revalidation. Datatypes may also define their own version types that aren't computed from the global clock; this helped us port preexisting concurrent code. Such values are also unordered and cause full revalidation.

Although predicates' covering version numbers aren't actually stored in the tracking set until the commit protocol begins, predicate evaluation nevertheless observes state, and whenever a predicate is checked, the corresponding version numbers must be checked for opacity. If an unordered version number is accessed in `check_predicate`, the containing transaction may spontaneously abort rather than risk observing inconsistent state.

STO's version of TL2 opacity is relatively fast, but its additional checks, and contention on the global version clock, introduce overhead. Some applications might prefer to avoid this overhead since they are written in a style that does not require opacity. Databases, for example, run user-defined code, and therefore already guard against infinite loops, traps, and so forth. For these applications, STO offers a mode in which opacity is turned off: opacity checks don't occur during transaction execution, and the global version clock is not used. A program whose transactions are safe, meaning a program whose transactions always abort or commit without invoking undefined behavior, crashing, or looping, may use the non-opaque mode in confidence. The original Silo database does not provide opacity, so we use the non-opaque mode for our Silo benchmark. We manually verified this was safe.

3.7 Implementation and discussion

Our implementation of STO has about 2,900 lines of C++ code in the core system, 6,600 lines in our datatypes, and many more lines of tests.

`TItem` efficiency was our major performance concern. The `Sto::item` function uses a per-transaction hash table to look up an item by owner and key, and $O(1)$ item access time is important for performance. Clearing the hash table after every transaction was expensive, so we designed a hash table that only needs clearing every several hundred transactions (depending on tracking set size). `TItems` are small, fixed-size objects; each one occupies four words (32 bytes). We considered alternate designs, including one where datatype implementers created subclasses of `TItem`. These designs avoided some pitfalls, such as type-safety issues with write values, but due to additional overhead, they performed badly.

Any state reachable from a `TItem` must stay in memory until the containing transaction commits or aborts. We ensure this by managing data structure memory using Read-Copy-Update (RCU) [19, 38] and making each transaction a read-side critical section. Thus, if transaction A observes a list node that transaction B deletes, then A's `check` method can safely access the node whether or not B has committed, since deleted nodes aren't freed until later.

Bounded spinning can cause false aborts, since a transaction waiting on an object locked by another transaction can abort even when there is no deadlock. Some datatype locks, such as those merely used to coordinate updates (and not used to detect conflicts), benefit from higher bounds.

Global-version-clock opacity is vulnerable to problems if version numbers wrap around, though this is rare given 64-bit version numbers.

The commit protocol calls `lock` and `install` callbacks in the order the corresponding `TItems` were added, and `unlock` and `cleanup` callbacks in reverse order. This makes it easier to communicate information from callback to callback, since one callback can set up object state for a later one to use.

4. Datatypes

We now turn to STO's transactional datatypes. We look first at how STO datatypes are designed in general, then describe those datatypes STO currently supports.

4.1 Methodology

This section describes methodological considerations, including how datatype specifications affect datatype implementation, several design patterns we have developed to handle common challenges, and some other issues that arise when implementing these types.

Specification A datatype's specification is crucial in determining how much potential concurrency it can support. In some cases, small changes to the specification can dramatically change the conflict graph [8]. If a hash table's `insert` method returns the new size of the hash table, then all insertions conflict with one another; changing the method to return void allows insertions of different keys to commute. The specification also constrains how the datatype's internal state can be divided into logical segments, which in turn affects how many version numbers a datatype contains and how many `TItems` its implementation will create. Programmers reason about logical segments by considering how operations commute. This reasoning process is common to many approaches to concurrent datatypes; for instance, the design process for a STO datatype's version numbers and `TItems` resembles the design process for a boosted data structure's abstract lock table [25].

Inserted elements Inserting a new item into a data structure is often a two-step process: the structure is searched for the right place to insert the item, and then the item is installed. Unfortunately, because of optimistic concurrency control, a naive STO datatype implementation might perform the expensive search step more than once per insert. In the transaction body, code would search for the key, but then record the new value in a `TItem` write value. During the commit protocol, the `check` callback would verify the absence of the key, and the `install` callback would again search for the insertion point to install the new value. This repeated searching can be expensive, particularly in $O(\log N)$ data structures like trees.

Many of our datatypes instead insert new items eagerly, during transaction execution. To preserve transactional correctness, these new items are marked as *poisoned*: any other transaction that observes the new item will abort immediately. This is an example of the direct update strategy for transaction execution [22], leading overall to STO using a hybrid strategy. Use of direct updates reduces lookups to the minimum of one per insertion, an important property for performance [48]. It also means transactional insertions need not be verified during the commit protocol: no other transaction can create a conflict with the inserted item since any transaction observing that item will abort.

Absent elements Absent elements require special care for transactional correctness. For instance, if a hash table's `get(K)` operation indicates that `K` is not in the table, a version number is required to verify this absence at commit time. Sensible storage locations for such version numbers vary from datatype to datatype. Our hash table, for example, uses a per-bucket version number that is incremented on insertion; our binary search tree uses version numbers on the absent node's possible parent(s). As a result, many of our transactional datatypes support several classes of `TItem`. Our hash table has `TItems` corresponding to present elements, to bucket versions, and to the hash table's size. These item classes are distinguished by using different portions of the key space, or by user flags on the items themselves.

The version numbers useful for detecting changes in absent elements have also proven suitable for validating range queries.

Read-my-writes Transactional correctness also requires that a transaction be able to read its own writes. Datatypes must check each `TItem` for a prior write and return the corresponding values if found (see Figure 2 for an example). They must also support combinations such as "a transaction inserts an item, then deletes it, then re-inserts it with a different value." When a data structure supports both direct updates (e.g., for inserted elements) and deferred updates, this creates complex interactions between direct updates and reads. For instance, consider a B+ tree transaction that first scans a key range, then inserts a key into that range. The direct-update insertion may change the versions observed by the range scan, but the transaction should still commit. Our datatypes handle this by updating the relevant items' version numbers when this occurs.

Correctness STO's correctness depends on datatype implementations following several rules. These include: (1) All accesses to shared state use version numbers: if transaction *A* makes a modification that transaction *B* observes, then some version number modified by *A* must have been read by *B* before *B* commits. (2) No two transactions can concurrently hold a lock on the same segment of state. (3) A `check_predicate` method must fail if the corresponding state has been modified in a semantically meaningful way. (4) A `check` method must fail if the corresponding segment is concurrently locked by another transaction or its version number has changed. (5) Data structure modifications must be invisible to other transactions before `install` is called. (6) Methods that acquire data structure locks must avoid deadlocks. Short-term spinlocks that are always acquired and released in the same method are generally fine; bounded spinning also works well.

Within these constraints, datatypes can implement their commit callbacks however they like.

Composition STO transactional objects can be composed. For instance, a transactional array can have transactional lists as elements. This works best given a layer of pointer indirection (array elements are *pointers to* transactional lists), since most transactional objects do not support operations like copying and assignment. Other composition strategies are future work.

Non-transactional operations Though transactional operations must be executed inside a transaction, datatypes are free to provide operations that execute outside of any transaction. Many of our datatypes do so, to support, for example, fast bulk load. Datatypes can also provide singleton concurrent operations that run outside of any transaction. This avoids transaction overhead and thus can perform somewhat faster, but such operations must perform sufficient version number updates to preserve the correctness of any concurrent transactions.

4.2 List of types

STO provides the following types in addition to `TBox<T>`.

`TCounter`. Like the counter discussed in §3.5, the library counter supports transactional increment, decrement, and test operations. Range-based predicates are used to reduce false conflicts: the C++ code `ctr > 2` observes a predicate on whether the counter's value is greater than 2, whereas a direct observation, such as `int i = ctr`, observes a specific counter value.

`TArray<T, N>`. This fixed-size array type offers transactional access to `N` indexed elements; the expression `a[i]` transactionally reads or writes element `i` as appropriate. Transactional iterators are also supported. For example, a transactional find of an element in an array can be implemented as follows:

```
extern TArray<T> arr;
TRANSACTION { ...
  auto it = std::find(arr.begin(), arr.end(), x);
... } RETRY(true);
```

`TArray`'s iterators follow the standard C++ iterator requirements, but are safe for use in transactions. Dereferencing an iterator acts like a read or write operation; for example, `*it = 3` becomes a transactional write to an element, and `z = *it` becomes a transactional read.

`TArray<T, N> x` behaves much like an array of transactional boxes `TBox<T> x[N]`, but there is a difference in overhead: the array of boxes contains `N` virtual function tables, one per box, while the array requires just one table.

`TVector<T>`. This datatype represents a variable-sized array. In addition to array access methods and iterators, it supports operations to change array size, including `push_back`, `pop_back`, `insert`, and `erase`. Thus, unlike in `TArray`, `TVector`'s size is a separately-observable logical segment. The underlying data is stored on the heap.

The standard C++ vector's `push_back` operation does not observe the vector's size, so two `push_back` operations in different transactions commute. Our implementation preserves this property: transactions that push elements onto a vector without observing the vector's size do not conflict. Furthermore, operations that access the size, such as iterator comparisons, `size`, and `empty`, use predicates to reduce false conflicts and reflect operation commutativity. The comparison `v.begin() == v.end()` merely observes whether `v`'s size is zero; the comparison `v.begin() + 3 < v.end() - 2` observes whether `v`'s size is greater than five.

`TList<T>`. This datatype implements a typed singly-linked list with a separate size component. Its main operations are `insert`, `erase`, `find`, and `size`. All updates to the list are serialized by a single lock (a potential area for improvement), but reads are lock-free thanks to RCU and version validation, and updates to different items commute. `TItem`s for lists correspond to list nodes, with a separate `TItem` representing the size.

`TListSet<T>`. This extension of `TList` implements a set (without duplicates); the type `T` must support equality, and the transactional `insert` method fails if the element is already present. Thanks to our insertion technique (direct updates), transactional insertions don't need commit-time verification.

`TQueue<T>`. Our transactional queue supports two operations, `push` (add element to tail) and `pop` (remove element from head). Like the `push_back` method in `TVector`, `push` does not actually observe the queue's state, so `push`-only transactions do not conflict.

`TPriorityQueue<T>`. Our transactional priority queue supports operations `push` (add element), `top` (return the top priority), and `pop` (remove a nondeterministically-chosen element with top priority). Our implementation is based on a concurrent max-heap stored in a C++ standard vector. All updates serialize at commit time on a coarse-grained lock, but their specifications allow many conflicts to be avoided: two pushes never conflict; two pops always conflict; and a push and a pop conflict only when the push adds a high-priority value. Our implementation takes advantage of this to substantial performance benefit (see §5.5). It also uses poisoning to encourage early aborts of conflicting transactions. For instance, a `pop`ped element is poisoned, causing concurrent `pop`ping transactions to abort. If the poisoning transaction aborts, the element's original version number is restored.

`THashtable<K, V>`. STO supports three map-like data structures, hash tables, binary search trees, and Masstree, a B+-like tree. The first of these, `THashtable`, is a concurrent hash table with chaining; it supports `get`, `insert`, `put`, and `remove` operations. Transactional inserts again use direct update; on commit, they update both the inserted element's version number and the version number on the corresponding bucket. When an element is found, the `get` method observes that element's version number; when not found, it observes the containing bucket's version number, ensuring that a later insert will abort the transaction. Removes are logically performed at commit time by marking the item as deleted (other transactions treat deleted items as absent). The actual unlinking, which requires a traversal, occurs during `cleanup`, after transaction locks are released.

`TRBTree<K, V>`. This datatype implements an ordered map using a binary red-black tree, and again supports `get`, `insert`, `put`, and `remove` operations. `TRBTree` supports both element version numbers, which are updated when tree contents change, and node version numbers, which are updated when tree *structure* changes, such as through nearby insertion or rotation. Element version numbers are used to validate successful gets and modifications, while node version numbers are used to validate unsuccessful gets (absent records). An unsuccessful get observes up to two version numbers, one for the node ordered immediately before the missing key and one for the node ordered immediately after it. If the missing key is inserted, at least one of the corresponding structural version numbers will change, even if the tree rebalances in the meantime. Transactional inserts again happen directly.

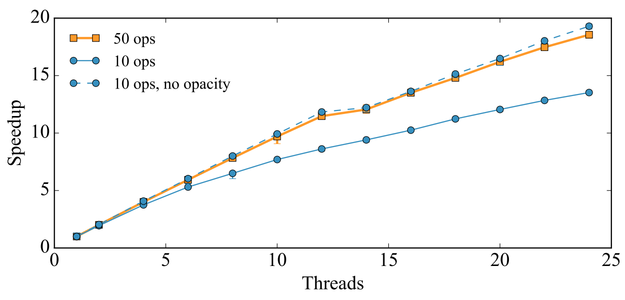

Figure 3. STO's scalability at low contention levels.

`TMasstree`. Masstree [36] is a high-performance concurrent B+ tree-like map data structure for key–value storage. `TMasstree` provides transactional access to a Masstree, with the same operations as `THashtable` plus a range query operation. Its `TItems` correspond to the values stored in the tree and, to support absent elements and range queries, tree leaf nodes. Only very small changes were required in Masstree to support transactional operation. Specifically, we changed Masstree to expose the previous version numbers of any leaf nodes that split as a result of an insertion. `TMasstree` uses this to update version numbers for prior range queries, as described in §3.7.

`TGeneric`. Finally, the `TGeneric` type implements an untyped STM: its `install` method makes changes to arbitrary words of memory. `TGeneric` both shows the generality of our system and is critical for supporting the STAMP benchmarks (in STAMP applications, some transactional accesses happen outside of any datatype). `TGeneric` stores version numbers by hashing memory addresses into a fixed-size array, while associating write values directly with destination addresses.

5. Evaluation

This section evaluates STO on microbenchmarks to demonstrate its scalability and the overhead of opacity; compares STO to traditional STM and transactional boosting using the STAMP transactional benchmark suite; and shows that STO can outperform a transaction system purpose-built for a large application by comparing it to Silo, a fast main-memory database.

Our experiments were run on a machine with two 6-core Intel Xeon X5690 processors clocked at 3.47GHz. The processors are hyperthread-enabled so there are 24 logical cores available. The machine has 100GB of DRAM in total, and runs 64-bit Linux 3.2.0. We compile STO with g++-5.3. In all graphs, we report the median of 5 consecutive runs, with the minimum and maximum shown as error bars (though often the error bars are too small to see).

```
intruder: -a10 -l2048 -n10000 -s1
genome: -g16384 -s64 -n16777216
kmeans: -m160 -n160 -t0.001 -i inputs/random-n65536-d32-c16.txt
labyrinth: -i inputs/random-x512-y512-z7-n512.txt
vacation: -n2 -q90 -u98 -r1048576 -t4194304
vacation-hi: -n4 -q1 -u90 -r184857 -t12194304
bayes: -v32 -r4096 -n10 -p40 -i2 -e9 -s1
```

Figure 4. Parameters used for STAMP benchmarks.

5.1 Microbenchmarks

We first evaluate STO's overhead and scalability limitations using microbenchmarks. Our test code performs transactions that read and increment random elements in a transactional array of 1 million integers. We measure both small (10-operation) and medium (50-operation) transactions; at this scale, contention is low (the abort rate is less than 1% even at 24 cores). In each transaction, half the operations are reads and half increments (so every operation reads an array index, with half of them then writing the same index). Arrays have particularly lean implementations in STO, so most of the overhead we see will come from the STO core shared by all datatypes.

Figure 3 shows the results. 50-operation transactions on STO scale well. Though scalability is not perfectly linear at larger core counts, much of the performance drop is due to memory accesses that cross the 12-core socket boundary. 10-operation transactions scale less well. This is due to the non-scalable global version clock STO uses to support opacity; with opacity disabled, 10-operation transactions scale almost exactly like 50-operation transactions. Disabling opacity has no noticeable performance impact on 50-operation transactions. These transactions take additional CPU time to execute, which reduces commits, and version clock updates, to a frequency well-supported by our hardware.

We also measured transactions accessing between 1 and 512 array elements with opacity disabled. Scalability is independent of size—different sizes' scaling curves are similar. Performance is not independent of size, however, since some per-transaction overheads are amortized at larger sizes. 128-item transactions commit 1.54x more items per second than 1-item transactions.

5.2 Typed transactions: STAMP

We now test our hypothesis that STO can outperform a conventional STM using STAMP, a transactional memory benchmark suite [39]. STAMP has been widely used to evaluate hardware and software transactional memories; it was designed for a conventional STM, and ships with a variant of TL2. Its component benchmarks model different parallel coding patterns. STAMP makes heavy use of datatypes such as lists, queues, and maps whose implementations call out to a word-based STM. We ported STAMP-0.9.10 to STO by adding STAMP-compatible interfaces to our transactional datatypes. Most STAMP benchmarks also contain some transactional accesses to other memory words, not part of any datatype; for those, we used STO's `TGeneric` implementation of word-based STM. We modified the suggested parameters for STAMP to ensure our workloads scaled to high core counts. At low core counts, we observe results with these parameters similar to those reported in the STAMP paper. Figure 4 gives our parameters.

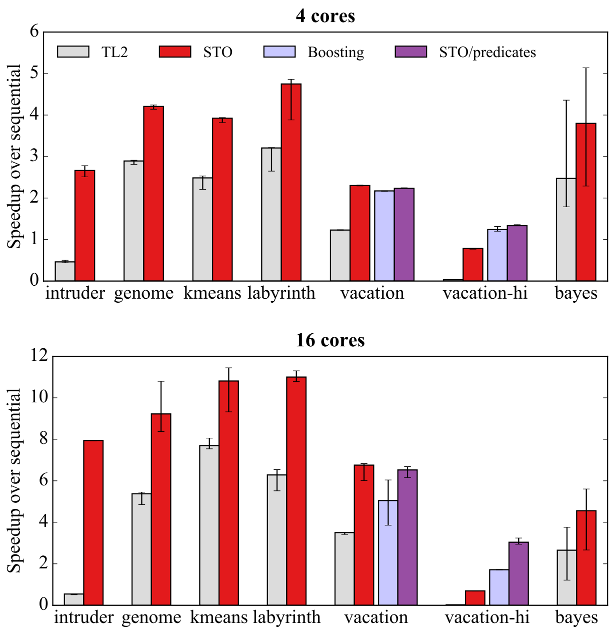

Figure 5. STAMP results: speedup on 4 and 16 cores over sequential code. Higher bars are better.

Figure 5 shows the results, at 4 and 16 cores, for six STAMP benchmarks (two others weren't suitable to port). Each bar shows the speedup obtained over STAMP's single-core version of the benchmark. The "Boosting" and "STO/predicates" results are discussed in the following sections. Some benchmarks exhibit superlinear speedup at 4 cores; reasons include benchmark nondeterminism (bayes [43, 44]), increased aggregate cache size (the cache miss rate in genome at 4 cores is 0.72x the 1-core value), and different datatype implementations (we follow STAMP's designs closely, but not religiously).

In this wide range of workloads, contention levels, and data structures, STO's speedup over sequential code always beats TL2's, often substantially. STO improves on TL2's speedup most significantly on the intruder benchmark. At 16 cores, STO's speedup is 14.6x higher than TL2's. Intruder uses list-sets heavily, and in word-based STMs, these sets add many extraneous words to tracking sets, such as for "next" pointers; the largest intruder transaction in TL2 has 40x more items than the largest intruder transaction in STO.

STO's benefits come from taking advantage of datatype semantics. We confirmed this by porting STAMP to use exclusively STO's `TGeneric`. The results were roughly comparable with TL2 on most benchmarks. An interesting exception was labyrinth, where at 16 cores, `TGeneric` outperforms

TL2 by 1.39x. This is due to our use of revalidation for opacity; in TL2, labyrinth has a high false abort rate.

We also measured SwissTM on these parameters [14], though using an earlier version of STAMP. SwissTM outperforms TL2, sometimes substantially, but STO outperforms SwissTM (by 1.1x–2.7x at 16 cores). Much of the benefit of SwissTM appears to come from its mixed-invalidation conflict detection scheme, which combines aspects of optimistic and pessimistic schemes. For instance, on vacation, SwissTM's performance resembles that of boosting (§5.3), which uses a pessimistic scheme; STO/predicates' application-defined conflict prevention (§5.4) performs much better than either. Broader adoption of SwissTM's mixed scheme in STO's datatypes might further improve STO's performance.

5.3 Boosting

We turn to a comparison with transactional boosting [25], another way to integrate concurrent datatypes into a transactional memory. Boosting has more inherent overhead than STO, namely lookups in a separate abstract lock table. (STO integrates transactional concurrency control into datatype memory layouts.) But since boosting uses 2-phase locking, it offers advantages at high contention. OCC generally outperforms pessimistic concurrency control when contention is low, but high abort rates at higher contention levels can cause its performance to collapse.

We evaluate boosting on STAMP's vacation benchmark. We implemented a version of boosting for generic map and list structures, and for vacation's application-specific reservation structure. (To validate this implementation, we ran it on the vacation parameters and core counts used by Herlihy and Koskinen [25]; the systems performed similarly.) Figure 5 shows the results (the "STO/predicates" bars are explained later). At low contention ("vacation"), boosting's overhead can reduce performance, and STO outperforms it by 1.4x at 16 cores. But things change at high contention. The "vacation-hi" benchmark has extremely high contention: transactions touch just 1% of the data. TL2 performs miserably. Its large read sets and unnecessary dependencies lead to very high abort rates and performance just 0.03x that of sequential code. STO's smaller read sets reduce transactional overhead substantially; it performs roughly 30x better than TL2, but still slower than sequential code. Boosting's pessimistic locking helps it perform roughly 2.5x better than STO, and even to scale, though not by much (1.72x at 16 cores).

But this tradeoff is not fundamental, since STO is flexible enough to support OCC, pessimistic concurrency control, and even boosting. To explore further, we created two new datatypes. Our *pessimistic* hash table obtains locks during operations and releases them in `cleanup` callbacks. Our *boosted* STO hash table achieves transactional correctness using a version of boosting. The boosted hash table is a wrapper around an unmodified concurrent hash table; the wrapper maintains an undo log and tracks abstract lock acquisitions using STO `TItems`, and its `cleanup` callback applies the undo log and releases locks as necessary.

STO	12.91x
Boosting	7.02x
Boosting in STO	6.67x
Pessimistic STO	9.82x

Figure 6. Effects of pessimism and boosting on a hash table microbenchmark. Numbers are speedup at 16 threads relative to single-threaded STO.

We evaluate these systems with a hash table microbenchmark. We run a series of transactions on a large hash table with 10 million possible keys. Each transaction performs 50 operations; every operation reads a value from the table, while 10% of the operations write a modified value back. The benchmark has low contention, so OCC should perform well. We measure four hash table implementations: STO's default optimistic hash table; boosting; the pessimistic STO hash table; and the boosted STO hash table. Figure 6 shows the results. STO OCC has the best speedup: the other systems' pessimistic locking imposes overhead, for instance to acquire read locks on read-only elements. Boosting and boosting-in-STO perform similarly, showing that STO's interface adds little overhead. But pessimistic STO outperforms the boosted variants. The difference is the extra overhead imposed by the abstract lock table inherent in boosting designs: every boosted hash table operation causes two lookups, one in the actual hash table being modified and one in the abstract lock table.

5.4 Application-defined operations

STO users can further improve performance by implementing their own transactional objects and specializing the commit protocol. We evaluated several such improvements in STAMP.

First, the kmeans benchmark performs a k-means cluster analysis on an array of multidimensional points. We initially followed STAMP and implemented cluster positions as transactional arrays, but then observed that each transactional update to a point changes all of its dimensions. It was faster to treat the point as a single object—a box containing a point, rather than an array of independent dimensions—since this reduced bookkeeping.

Second, the vacation benchmark models complex reservations across multiple classes of travel goods (cars, hotel rooms, flights). Each class has a price, a total count, and a count of goods available. By default, two reservations for the same class (e.g., two room reservations in hotel #3) will always conflict, even though they semantically commute when enough goods are available (e.g., two or more available rooms in hotel #3).

We initially used a transactional box for each class of goods, but observing many false conflicts, we upgraded to a custom transactional datatype. Predicates are used to implement requirements like "reserve if a good is available," "test if a good is available," and "cancel an outstanding reservation if one exists," avoiding many false conflicts when concurrent transactions access the same class. Predicating the class-of-goods datatype added about 80 lines of code. The "STO/predicates" line in Figure 5 shows the results. Under low contention, STO/predicates performs similarly to STO. Under high contention, however, application-specific conflict detection shines. STO/predicates preserves a relatively low abort rate, and at 16 cores, it outperforms STO by 4.3x and boosting by 1.8x.

5.5 Datatype design

In this section we briefly explore how our standard datatypes benefit from STO's ability to manage conflicts. We center on our `TVector` datatype, which implements an array whose size can change.

In our initial `TVector` implementation, operations such as `push_back`, `pop_back`, and `end` observed the vector's size, leading to high conflict rates in experiments with concurrent pushes and pops. Predicates enabled a better implementation, in which, for instance, `pop_back` observes only whether the vector's size is greater than 0, and an iterator comparison like `v.end() - v.begin() < 5` observes whether the size is less than 5. The resulting reduction in conflicts can greatly improve performance. We ran a microbenchmark consisting of randomly-chosen 4-operation transactions; each operation is either a size comparison, a `push_back` operation, a `pop_back` operation, or a `find` operation using iterators. At 16 cores, our predicated `TVector` implementation has 1.5x higher transaction throughput than the non-predicated implementation.

Second, we turn to standard algorithms. C++ offers a priority queue container (`std::priority_queue`) that adapts any vector-like datatype into a max-heap-based priority queue. `TVector` is suitable for this adapter, so `std::priority_queue<X, TVector<X>>` is a correct transactional priority queue. This ease of programming is an advantage of the STO implementation style. Unfortunately, the adapter implementation incurs high levels of conflict, since `std::priority_queue`'s heap algorithms observe much of the vector's internal state. From the vector's point of view, these observations cause true conflicts: it is only when considered relative to the priority queue's specification that the conflicts are false. Our purpose-built priority queue avoids these false conflicts. The implementation cost is several hundred lines of code, but the performance benefits are substantial. We ran a microbenchmark in which 75% of transactions push one random value onto a shared priority queue and the other 25% pop its top three values. At 16 cores, the specialized priority queue commits 2.3x more transactions per second than the adapter.

5.6 Silo

Finally, we show that despite its generality, STO can outperform a transaction implementation purpose-built for a large application. Our comparison system is Silo, a fast in-memory database [48]. Silo implements transactions using Masstree; it adds transaction-aware record structures, an optimistic concurrency control-based transaction protocol, and an implementation of TPC-C, the well-known high-performance transaction processing benchmark [47]. The speed of Silo transactions inspired our work on STO: Silo outperforms many other in-memory databases by orders of magnitude. To adapt Silo to STO, we replaced Silo's commit protocol and tree interface with wrappers around STO's `TMasstree` datatype, and replaced the transaction-aware record structure with a STO variant; only the TPC-C implementation was left unchanged.

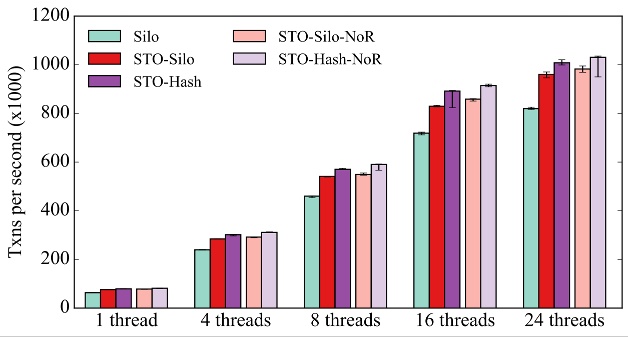

Figure 7. Results in total transactions per second for TPC-C (standard mix) for Silo, an in-memory database built using ad-hoc concurrency, and two versions of STO-enhanced Silo, one that uses `TMasstree` for all tables (STO-Silo) and another that uses `THashtable` instead of `TMasstree` for some tables (STO-Hash). In "STO-Silo-NoR" and "STO-Hash-NoR", we disable read-my-writes (Silo also disables these features). Silo is very fast, but our STO-Silo version outperforms it by 1.17x and our STO-Hash version outperforms it by 1.23x.

We measure Silo and STO-Silo on TPC-C on 24 cores. Our experiments run for 30 seconds; Silo's logging and persistence support is disabled (so our results correspond to "MemSilo" in Tu et al. [48]), and STO's opacity support is disabled. Figure 7 shows the results. (Though TPC-C performance is often measured as New Order transactions per minute, the figure follows Silo and reports total transactions per second for the standard TPC-C mix.) STO-Silo performs well; in fact, at 24 cores, it has 1.17x higher throughput than Silo.

STO-based Silo is both simpler and more general than the original. The STO core plus `TMasstree` is less than 3,000 lines of C++ code, while the corresponding part of Silo is more than 7,000 lines long. Furthermore, whereas Silo's implementation is deeply enmeshed with Masstree, STO makes it easy for STO-Silo to try different datatypes. Some of the TPC-C benchmark's tables are never used in range queries, and thus can be implemented as hash tables, avoiding the overhead of $O(\log N)$ tree lookups. The STO-Hash system in Figure 7 updates STO-Silo to use `THashtable` for five of its tables. STO made this a simple change. STO-Hash performs even better, with 1.23x Silo's throughput at 24 cores.

Moreover, all these comparisons are somewhat unfair to STO. Silo's transaction system doesn't support reading an item that was written by the same transaction (TPC-C does not require this). The NoR variants in Figure 7 modify STO in a similar way; these variants perform even faster. Although we confirmed that opacity is not necessary for the TPC-C benchmark, we also ran the benchmark with STO's opacity enabled, and observed only 5% overhead.

Our original implementation of STO was only slightly faster than Silo. However, we found that STO's performance on TPC-C improved over time, as we optimized STO for other, unrelated benchmarks. In particular, our hash table to provide $O(1)$ lookup of `TItems`, and our choice not to sort the write set, gave a nearly 10% combined improvement on TPC-C. Given more effort, a purpose-built system for Silo transactions could likely outperform our general system. Our results show, however, that such an effort might not be worthwhile. STO's generality has low cost, and further improvements on STO's transactional core would benefit not only STO-Silo, but also other programs.

6. Conclusions

STO is a new software transactional memory that derives power from abstraction. Rather than concrete reads and writes to memory words, STO manages abstract read and write operations on transactional datatypes. It then delegates much of its commit protocol to datatype callbacks. STO datatypes can implement their own concurrency control mechanisms, thereby reducing false conflicts and improving scalability. They can also decouple their internal concurrency control from transaction processing. New commit protocol features, including optimistic transactional predicates, allow datatypes to capture much of the concurrency latent in their specifications. STO's transactional core is highly efficient despite its abstract nature. Our large datatype library demonstrates a range of implementation techniques, and proves STO to be an effective framework for transactional datatype programming. STO outperforms both TL2, a conventional software transactional memory, and transactional boosting. More significantly, it outperforms Silo, a purpose-built transaction system, on the TPC-C database benchmark.

In the future, we hope to apply STO to more concurrent programs and to explore extensions of its approach to persistent and distributed transactions. We also hope to implement improvements including more scalable version clocks, mixed invalidation, and more mechanisms for reducing false conflicts, such as transactional futures [41].

STO is available at github.com/nathanielherman/sto.

Acknowledgements

We thank our shepherd, Madan Musuvathi, and our Eurosys anonymous reviewers for their helpful comments. We are also grateful to Min Zhang (MIT) for implementing an initial version of STO's application-specific concurrency control for vacation. Thanks to the Harvard College Research Program for supporting some of Nathaniel Herman's work. This material is based upon work supported by the National Science Foundation under Grant Nos. 1513416 and 1302359. Eddie Kohler was partially supported by a Microsoft Research New Faculty Fellowship.

References

[1] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, Mar. 1992.

[2] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: High-performance concurrent sets and maps for STM. In *Proc. PODC '10 (29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing)*, pages 6–15. ACM, 2010.

[3] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63:172–185, Dec. 2006.

[4] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A hybrid transactional memory for Haswell's Restricted Transactional Memory. In *Proc. PACT '14 (23rd International Conference on Parallel Architectures and Compilation)*, pages 187–200. ACM, 2014.

[5] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proc. PPoPP '07 (12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming)*, pages 56–67. ACM, 2007.

[6] F. M. Carvalho and J. Cachopo. STM with transparent API considered harmful. In *Proc. ICA3PP '11 (11th International Conference on Algorithms and Architectures for Parallel Processing)*, LNCS 7016, Melbourne, Australia, Oct. . Springer.

[7] C. Caşcaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Communications of the ACM*, 51(11), Nov. 2008.

[8] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proc. SOSP '13 (24th ACM Symposium on Operating Systems Principles)*, Farmington, PA, Nov. 2013.

[9] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proc. PPoPP '10 (15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming)*, pages 67–78. ACM, 2010.

[10] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory.

SIGARCH Computer Architecture News, 39(1):39–52, Mar. 2011.

[11] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proc. ASPLOS '06 (12th International Conference on Architectural Support for Programming Languages and Operating Systems)*, pages 336–346. ACM, 2006.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. DISC '06 (20th International Conference on Distributed Computing)*, Stockholm, Sept. 2006.

[13] A. Dragojević and T. Harris. STM in the small: Trading generality for performance in software transactional memory. In *Proc. EuroSys'12 (7th European Conference on Computer Systems)*, Bern, Switzerland, Apr. 2012.

[14] A. Dragojević, R. Guerraoui, and M. Kapałka. Stretching transactional memory. In *Proc. PLDI '09 (ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation)*, Dublin, June 2009.

[15] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of ACM*, 19(11):624–633, Nov. 1976.

[16] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proc. DISC '09 (23rd International Conference on Distributed Computing)*, pages 93–107, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] S. M. Fernandes and J. Cachopo. A scalable and efficient commit algorithm for the JVSTM. In *Proc. TRANSACT 2010 (5th ACM SIGPLAN Workshop on Transactional Computing)*, Paris, Apr. 2010.

[18] S. M. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proc. PPoPP '11 (16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming)*, San Antonio, TX, Feb. 2011.

[19] K. Fraser. *Practical Lock-freedom*. PhD thesis, University of Cambridge, 2004.

[20] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic scalable atomicity via semantic locking. In *Proc. PPoPP '15 (20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming)*, pages 31–41. ACM, 2015.

[21] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proc. PPoPP '08 (13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming)*, pages 175–184. ACM, 2008.

[22] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[23] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *Proc. PPoPP '14 (19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming)*, pages 387–388, Orlando, FL, 2014. ACM.

[24] A. Hassan, R. Palmieri, and B. Ravindran. On developing optimistic transactional lazy set. In *Proc. OPODIS '14 (18th International Conference on Principles of Distributed Systems)*, LNCS 8878, pages 437–452, Cortina d'Ampezzo, Italy, 2014. Springer-Verlag.

[25] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proc. PPoPP '08 (13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming)*, pages 207–216. ACM, 2008.

[26] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA '93 (20th Annual International Symposium on Computer Architecture)*, pages 289–300. ACM Press, 1993.

[27] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[28] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. In *Proc. PODC '02 (21st Symposium on Principles of Distributed Computing)*, pages 131–131. ACM, 2002.

[29] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proc. PODC '03 (22nd Annual Symposium on Principles of Distributed Computing)*, pages 92–101. ACM, 2003.

[30] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.

[31] H. F. Korth. Locking primitives in a database system. *Journal of ACM*, 30(1):55–79, Jan. 1983.

[32] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proc. PLDI '07 (ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation)*, pages 211–222, San Diego, CA, 2007. ACM.

[33] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *Proc. PLDI '11 (32nd ACM SIGPLAN Conference on Programming Language Design and Implementation)*, pages 542–555, San Jose, CA, 2011. ACM.

[34] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[35] M. Lesani and J. Palsberg. Decomposing opacity. In *Proc. DISC '14 (28th International Conference on Distributed Computing)*, 2014.

[36] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys'12 (7th European Conference on Computer Systems)*, Bern, Switzerland, Apr. 2012.

[37] A. Matveev and N. Shavit. Reduced hardware NOrec: A safe and scalable hybrid transactional memory. In *Proc. ASPLOS '15 (20th International Conference on Architectural Support for Programming Languages and Operating Systems)*, pages 59–71, Istanbul, 2015. ACM.

[38] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proc. PDCS '98 (10th IASTED International Conference on Parallel and Distributed Computing and Systems)*, pages 509–518, Las Vegas, NV, October 1998.

[39] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-

Processing. In *Proc. IISWC '08 (4th International Symposium on Workload Characterization)*, 2008.

[40] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proc. PPoPP '07 (12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming)*, pages 68–78, San Jose, CA, 2007. ACM.

[41] L. Pina and J. Cachopo. Profiling and tuning the performance of an STM-based concurrent program. In *Proc. TMC '11 (Workshop on Transitioning to MultiCore, part of SPLASH '11, ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity)*, Portland, OR, Oct. 2011.

[42] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proc. DISC '06 (20th International Conference on Distributed Computing)*, pages 284–298, Stockholm, 2006. Springer-Verlag.

[43] W. Ruan and M. Spear. Hybrid transactional memory revisited. In *Distributed Computing*, pages 215–231. Springer, 2015.

[44] W. Ruan, Y. Liu, and M. Spear. STAMP need not be considered harmful. In *TRANSACT'14 (9th ACM SIGPLAN Workshop on Transactional Computing)*, 2014.

[45] P. M. Schwarz and A. Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2 (3):223–250, Aug. 1984.

[46] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proc. SSS '11 (13th International Conference on Stabilization, Safety, and Security of Distributed Systems)*, pages 386–400, Grenoble, France, 2011. Springer-Verlag.

[47] The Transaction Processing Council. TPC-C benchmark (revision 5.9.0). http://www.tpc.org/tpcc/, June 2007.

[48] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. SOSP '13 (24th ACM Symposium on Operating Systems Principles)*, pages 18–32. ACM, 2013.

[49] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computing*, 37(12): 1488–1505, Dec. 1988.

[50] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *Proc. PPoPP '15 (20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming)*, pages 76–86. ACM, 2015.

[51] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel Transactional Synchronization Extensions for high-performance computing. In *Proc. SC'13 (International Conference for High Performance Computing, Networking, Storage and Analysis)*, Denver, CO, Nov. 2013.

[52] M. Zhang, J. Huang, M. Cao, and M. D. Bond. Low-overhead software transactional memory with progress guarantees and strong semantics. In *Proc. PPoPP '15 (20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming)*, pages 97–108. ACM, 2015.