Memory Abstractions for Data Transactions

Memory Abstractions for Data Transactions

by

Nathaniel Herman

Submitted to the Department of Computer Science
on April 1, 2015, in partial fulfillment of the
honors requirements for the degree of
Bachelor of Arts in Computer Science
at Harvard University

Abstract

This thesis presents STO, a software transactional memory (STM) based not on low-level reads and writes on memory, but on datatypes—arrays, lists, queues, hash tables, and so forth—that explicitly support transactional operations. Conventional STMs allow programmers to write concurrent code in much the same way as sequential code—thereby more easily taking advantage of multiple CPU cores. However, these conventional STMs track every memory word accessed during a transaction, so even simple operations can perform many more memory accesses for synchronization than is strictly required for transactional correctness. Our insight is that concurrent data structures can generate fewer superfluous accesses, and use more efficient concurrency protocols, when transaction bookkeeping tracks high-level operations like "insert node into tree." We test our ideas on the STAMP benchmark suite for STM applications, on a high-performance in-memory database, and on a previously single-threaded program that we extend to multithreaded operation. We find that datatypes can support transactional operations without too much trouble; that relatively naive users can build simple transaction support into their own data structures; and that our typed STM can outperform and outscale conventional, untyped STM, and even outperform world-class, hand-rolled transactional implementations for databases. This thesis also includes a full implementation of the system, which will be made open source and is currently available on request.

Thesis Supervisor: Eddie Kohler
Title: Professor

Acknowledgments

A huge thank you to Eddie Kohler, my advisor, mentor, and friend. He got me involved in systems research in the first place, first convincing me to add numbers with an image of a German train map (if that can be called research), then to take his systems research class, and then he even convinced me to forgo a tech internship in favor of the summer research that has turned into this thesis. He might even be able to convince me to go to graduate school next. Besides changing my mind on things, Eddie has been a great help in brainstorming hard problems, helping me get unstuck, and changing the way I think about systems.

I am also very grateful to my other readers, Margo Seltzer and Jim Waldo. I have learned lots and become a much better programmer through Professor Seltzer's classes, and she has always been very open and helpful in my research endeavors. Professor Waldo, who is also my academic advisor, has given me lots of helpful advice over the years, and is always fun to talk to.

This thesis is based off of a submission to the 25th ACM Symposium on Operating Systems Principles (SOSP), and thus represents joint work done with Eddie Kohler, Barbara Liskov, Jeevana Inala, Yihe Huang, Liuba Shrira, Min Zhang, and Lily Tsai. I would of course like to thank all of my collaborators for their help on the project; they did great work and were lots of fun to work with. My summer research was supported by the Harvard College Research Program, which I am also very grateful for.

Finally, I'd like to thank my family, including my parents, Kristine and Steve Herman, and my sister Natasha Herman, for all of their support over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Concurrent programming (programming involving multiple computations run in parallel) is both notoriously difficult and extremely important. For many years, processor performance improvements meant yesterday's programs ran much faster on new hardware with no changes required. More recently, though, processor improvements come in the form of parallelism: a computer's processing power improves in the form of more processors (cores), rather than faster processors. Yesterday's programs must be rewritten to take advantage of these new resources. Yet many new programs are still written in the traditional, sequential style, failing to take advantage of concurrency. The reason is simple: programming multiple computations to interact together significantly increases complexity and makes programs harder to reason about.

Unfortunately, methods for handling concurrency are often low-level and require significant implementation overhead to use in a system efficiently. Thus, it is desirable to have an integrated method for handling concurrency, one that efficiently implements the necessary low-level complexities for the programmer automatically. One such method for dealing with concurrency is that of transactions. A transaction is a list of operations that together have an "all-or-nothing" property—either all of them are performed (in a consistent state) or none of them.

It is no coincidence that transactions bring to mind banking: indeed, one use for a transaction is in implementing a banking transaction. If your account has an initial balance of $100, and you deposit $100 at the same time that your work deposits a paycheck of $100, what should your final balance be? A naive implementation of banking code would make a deposit by reading your current balance, adding $100 to it, and then setting that as the new balance. However if, between reading

the current balance and adding $100, the paycheck was deposited, this method would lead to a final balance of $200, instead of $300. The bank might not mind this scenario, but you would not be so happy. However, if the read and write operations were instead wrapped in a transaction, the transaction would fail (and then retry) in this scenario, since the balance changed mid-transaction.

Transactions were originally designed for databases, which have been handling concurrency for decades. But transactions also offer hope for general programmers. Transactions appear to the programmer as if they execute one at a time, in series, with no interference from other concurrently running code. This is exactly the guarantee that sequential programs expect! Thus, transactional memory (TM)—which employs transactions on memory, rather than a database—provides a method to speed up general programs while still allowing programmers to write code very similar to sequential code.

1.1 Transactional Memory

Transactional memory works by allowing programmers to delimit blocks of code ("transactions") that should appear to execute atomically and in isolation. Committed transactions have *serializable* effects: the results of any transactional program must be equivalent to the results of some execution where the committed transactions ran one after the other (serially), with no interference from other transactions. It is this restriction on visible effects that makes transactional programming simpler for users.

A transaction contains normal program statements (including control flow statements, for instance) referring to a mixture of global and local state. A TM user (or compiler) must ensure that all global accesses are routed through the transaction system. This system tracks read and written objects, watches for conflicts, and ensures that all changes to global state commit as an atomic unit. Importantly, although each transaction *appears* to execute one after the other, in reality transactions run in parallel, and it is the TM system that ensures that these parallel executions can be reconciled into an equivalent serial execution.

TM transactions can express many concurrent algorithms; naturally provide fine-grained concurrency; avoid issues that plague ad hoc parallel programming, such as deadlock; and unlike techniques such as locking, transactions compose naturally. TMs were designed for hardware imple-

mentation [22], and recent Intel Haswell and IBM processors have hardware TM (HTM) support. But HTMs cannot support all transactions. There are capacity issues—current implementations rely on the cache coherence protocol, and thus require all memory accessed inside a transaction to fit within, e.g., an L1 cache—and other microarchitectural issues can prevent a valid transaction from ever committing [36]. An HTM is thus generally backed up by a software TM (STM), such as TL2 [12]. STMs implement all the necessary machinery for transactions entirely in software.

Many STM transaction systems use optimistic concurrency control (OCC) [27]. A library tracks all read objects in a *read-set* and all written objects in a *write-set*. To commit a transaction, the system executes a multiphase *commit protocol* to ensure serializability. In Phase 1, the system *locks* all memory words in the write-set. This prevents other concurrent transactions from accessing the entities modified by the current transaction. Some technique, such as lock ordering or bounded spinning (aborting after a fixed period of unsuccessfully acquiring a lock), is required to avoid deadlock. In Phase 2, the system *checks* all words in the read-set; if a word changed since its original access, or if the word is locked by another transaction, the transaction must abort. The commit point for a transaction is immediately after Phase 2 successfully completes. In Phase 3, the system *installs* the modified words, which overwrites memory locations with the values logged in the write-set. Finally, the system *unlocks* the write-set (this must happen whether the transaction commits or aborts).

STM systems must provide both a lock and a *version* for each shared-memory word used in a transaction. A word's version (typically an integer) is updated every time the word is modified by a successful transaction. The read-set contains versions, not memory values, and the Phase 2 check verifies that a word's version has not changed since it was read; this catches serializability violations that simple value comparison would miss (known as the ABA problem [11]). Various techniques can be used to optimize lock and version storage.

STMs avoid the size restrictions of HTMs, but at a performance cost. Although there are no microarchitectural restrictions on STMs, large read- and write-sets can still cause performance problems, and even fast STM implementations significantly lag the performance of purpose-built concurrent code [7].

1.2 STO

Our work attempts to make a software transactional memory faster by, perhaps paradoxically, making it more general. We call our design *software transactional objects*, or *STO*. STO provides what one might call a *typed* STM. Rather than mimicking the restricted, untyped-memory-based designs of hardware transactional memories, STO is designed primarily for software implementation. STO tracks global accesses *abstractly*, at the level of datatype operations, rather than concretely, at the level of memory words. For instance, consider a transactional `find` operation that looks up a value in a transactional sorted list. A typed STM can track this operation using a single abstract read-set item that represents this exact operation. An untyped STM, in contrast, would store a read-set item for every shared-memory word accessed during the operation; the expected number of items will be proportional to the length of the list. This not only adds overhead, it also increases the likelihood of false conflict. In a typed STM, an insertion of a different item into the list need not invalidate the `find`, while in the untyped STM, many such insertions will do so.

STO *transactional datatypes* (i.e., datatypes that export transactional operations) share the responsibility of providing transactional correctness with the transaction system. A transactional datatype provides transactional operations called by users, and a handful of methods called at commit time by the transaction system. The transactional operations modify the current transaction to track abstract reads and writes, while the commit-time methods verify these reads and install the writes in a manner appropriate for the datatype. The author of a transactional datatype must leverage the transaction system's primitives to provide transactional semantics, and must implement concurrency control on the shared-memory data structure. Usually concurrency control is required in both the transactional operations (e.g., the list's transactional `find` operation) and the commit-time methods (e.g., the code that installs an inserted item into the list). Concurrency control is never an easy task, but in our experience, transactional methods on a concurrent datatype are not too much more difficult to program than correct non-transactional concurrent methods. Our design also gives datatype designers maximum freedom: there are very few restrictions on the kinds of concurrency control a datatype may use. (In fact, a datatype could use *hardware* TM in its operations!) This freedom offers designers opportunities to improve performance further. For example, co-designing a list's concurrency control protocol with its transactional operations allowed us to

implement a transactional insert operation that adds *no* operations to a transaction's read set, so that insertions never conflict. Although the typed STM design does complicate the task of datatype designers, *using* a transactional datatype in STO is no more difficult than using a similar datatype in an untyped STM. In fact, user code can be almost identical.

We describe the STO design and its implementation. STO is a C++ library comprising a small core transactional system and several transactional datatypes, including lists, hash tables, a B+ tree variant, and a generic "box" wrapper that can make reads and writes of any datatype transactional. The main challenge of the design was to achieve generality, and maximize usability, without sacrificing high performance; we explain the tradeoffs required. We evaluate STO on microbenchmarks, the STAMP TM benchmark suite, and on a challenging macrobenchmark, the Silo high-performance main-memory database [34]. STO scales well and can exceed the performance of the fast TL2 software transactional memory by anywhere from 1.2-5x on STAMP benchmarks. Our reimplementation of Silo exceeds Silo's performance by 20% on the TPC-C benchmark, even though Silo was heavily optimized; also our reimplementation contains far less code and is easier to read. Finally, we evaluate effectiveness for general systems code by porting the thttpd single-threaded web server [1] to STO. A few transactional annotations suffice to make this server correctly use multiple cores. Our main contribution is the STO design for a typed software transactional memory. Our secondary contributions include the implementation and our faster, simpler Silo design.

The remainder of this thesis consists of 7 chapters. In chapter 2, we describe related work to STO; in chapter 3, we provide an overview of our system. In chapter 4, we further expand on STO's design, providing examples of transactional datatypes and how they are used. In chapter 5, we discuss implementation details of the system, and in chapter 6 we describe the many datatypes we have designed for STO. In chapter 7, we evaluate our system on microbenchmarks, on the popular STAMP benchmarks, against the Silo in-memory database, and finally perform a case-study of converting an existing single threaded program into a multithreaded one using STO. In chapter 8, we discuss future work and conclude.

Chapter 2

Related Work

Support for transactions can be implemented in hardware or software. Support for hardware transactions (HTM) [22] is available in recent Intel [25] and IBM [2] processors, with others likely to follow suit. Hardware transactions have low cost but for the foreseeable future they will not replace software approaches because HTM is resource constrained. This makes hardware transactions not truly composable [17].

Software transactional memory (STM) [8, 12, 24, 37] relaxes the physical resource limitations of HTM. It can support arbitrary size transactions at the cost of performance overhead. The performance overheads of software transactional memory are due to the high cost of single thread overheads to track conflicts at memory word granularity, aborts due to conflicts, and the cost of consistency checks.

Early techniques to speed up transactions by eliminating false conflicts (e.g. open nesting [32], elastic transactions [13], transactional collection classes [6], and early release [23]) expose non-transactional APIs, making them hard to use by ordinary programmers.

Object-based STMs [14, 24] work at the object level rather than the word level. However, unlike STO these STMs make shadow copies of any objects written to, requiring both excessive copying and no reduction of false conflicts.

Our work most relates to prior research that speeds up transactions using abstraction, including boosting [21], optimistic boosting [20], automatic locking [15] and predication [3]. Like our approach, these techniques rely on highly-concurrent data structures written outside the STM and expose a simple transactional API to ordinary programmers. Relying on data structures allows a

reduction of the single thread overheads that limit scalability even in the current best performing Lark STM [37]. The main difference between these approaches and STO is that the approaches center on adapting data structures to a concrete untyped memory-word-based STM. Our work, in contrast, flips the design: datatypes are primary, and our typed STM is abstract. Despite this change, our approach offers performance advantages beyond those reported for prior systems. We now consider these approaches further.

Boosting [21] wraps unmodified highly-concurrent data structure implementations for use by STM transactions. The wrapper exploits operation commutativity, avoiding false STM conflicts, and performs undo logging (to roll back any changes made by a transaction that aborts). Automatic locking [15] synthesizes the semantic locking scheme automatically from commutativity specifications. In these locking based schemes all accesses must first acquire a semantic lock, which penalizes read performance. Optimistic boosting [20] reduces the cost of semantic locks by replacing them with lazy optimistic locks, logical redo logging, and flexible optimistic validation, but this approach is less general than ours and experimental results show small performance gains.

Predication, a technique specialized to sets and maps, couples operations on concurrent data structures with operations on a table of STM "predicate words," where each predicate word corresponds to a semantic property of the underlying data structure (such as the absence of a key). A lookup of an absent key inserts a predicate word, introducing a difficult garbage collection problem. STO's typed transactions naturally avoid this problem.

Modern support for transactions provides opacity [16, 28], a condition that guarantees the code using transactions observes a transactionally consistent memory state even when transactions abort. Checking opacity can be costly and STM systems use efficient imprecise techniques for preserving opacity. TL2 [12] uses a global version clock, while NOrec [8] combines a global counter with value based validation. Like NOrec, STO uses efficient imprecise checks, treating them as hints to trigger a precise detection; however, our precise checks are semantic rather than based on values, which allows us to avoid unnecessary aborts.

Hybrid approaches [5, 9, 10, 35] reduce the performance overheads of STM by combining hardware and software techniques. Recent 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 consider STO combined with HTM to be future work.

Chapter 3

Overview of approach

3.1 Typed transactions

The central idea of STO is that a transactional memory can, and should, track *semantic* reads and writes, using datatype-specific implementations of the actions required by the commit protocol (lock, check, install, unlock), rather than tracking concrete memory words. For example, a transactional tree's `find(key k)` operation might track only the leaf node where `k` resides, avoiding any concern about modifications that affect the path to that key. Similarly, a transactional array can track accesses by index rather than memory location, allowing it to move its storage without causing aborts. Transactional datatypes can ensure that semantically independent accesses never appear to conflict, even if they modify the same memory; for example, distinct inserts to a data structure representing a set might modify the same internal list, but they don't need to conflict at the transaction level. A way to understand the system is that we allow datatype implementers to co-design the short term concurrency control that is used to execute a transactional operation (e.g., a transactional insert into a set) and the transactional concurrency control that comes into play later when the transaction attempts to commit.

Implementing STO's generality inevitably adds overhead; the commit protocol now performs method calls on transactional datatypes, and generality leaves fewer opportunities for system-wide optimization. Our hypothesis is that the advantages of typed transactions can outweigh this overhead. Those advantages are as follows.

Smaller read- and write-sets. In all but the simplest datatype, any operation requires many

accesses to shared memory. For example, in a balanced tree structure, finding an item with a given key will touch words at many internal nodes. An untyped STM would log each of these words, but STO can log a higher-level, *semantic* item representing the found key. Our evaluation shows that STO can in some cases track hundreds of times fewer items than an untyped STM. Besides the more obvious benefits from tracking fewer items (less memory and less time spent searching and validating items), tracking fewer items can also reduce conflicts. Phase 2 (validating reads) must occur after Phase 1, that is, after any writes for the transaction have been locked. Larger read-sets mean more time spent in Phase 2, meaning more time spent holding locks, which both locks out other writers and invalidates more reads.

Fewer false conflicts. Tracking semantic accesses rather than word accesses also avoids false conflicts. Returning to the balanced-tree example, internal nodes are subject to modification when the tree rebalances, whether or not the found item changed. STO datatypes can track more precise conflict information, and in particular can add items to the read-set only when they could affect the high-level result of a transactional operation. In our tree type, for example, rebalances need not cause aborts.

Datatype-specific concurrency control. Datatypes must implement concurrency control on their in-memory structures; e.g., a transactional list must use local spinlocks so that concurrent insertions don't violate structural invariants. Though this requires implementation work by datatype designers, it also means the transaction system as a whole can use any concurrency control mechanism a designer can think of. Datatypes can implement their own locks, which need not align with the logical locks used in the commit protocol. They can also use advanced techniques like version validation [4] or lock-free instructions, or even use hardware transactional memory. Since the best concurrency control mechanism for a data structure often depends intimately on that structure's layout and operations, this offers flexibility an untyped STM cannot match.

Conflict elimination. Finally, datatypes can design their concurrency control mechanisms (e.g., the locks used when looking up an item) to reduce, or even eliminate, entire classes of false conflict. For instance, in STO insertion of a new item into a sorted list actually inserts a placeholder item *at execution time*, before the transaction commits. Execution-time concurrency control ensures that at most one concurrent insertion of the same item will succeed; other transactional operations ignore the placeholder or abort on encountering it as appropriate. This design means that

successful transactional insertions are *conflict-free*: no concurrent transaction can invalidate such an insertion, and the insertion need not add any items to the transaction's read-set. Concurrent inserts can therefore have zero false conflicts and zero checking.

Of course, typed transactions have disadvantages as well. Datatype authors must implement transactional operations, and users must call transactional methods. We address these questions further below.

3.2 Target programmers

We aim to provide a new way to support efficient concurrent programming. Our goal is to provide both an extensible library of fast transaction-aware concurrent datatypes, and a system allowing relatively naive users to write arbitrary transactions using those types.

An important point is that most programmers will not need to implement transactional datatypes, since they can use ones available through libraries. We hope that our mechanisms will make it easy for end-user programmers to write transactional programs. These programs should both avoid bugs and perform well. We describe several mechanisms, such as transactional boxes (§6), meant to simplify these programmers' tasks. On the other hand, the design of highly-efficient concurrent data structures requires sophistication. We do not expect it to be easy for datatype programmers to support transactions, though we believe our approach does not add too much difficulty over the fundamental problem of correct concurrent programming. Still, our target datatype programmer must be more sophisticated than end-user programmers of our system. This idea generally holds true in programming, as library programmers tend to require more sophistication than the users of these libraries.

Chapter 4

Design

This section describes the design of STO's typed transaction system. We explain how STO transactions are used and how they are implemented, mention important optimizations, and describe how we implement opacity (the property that a transaction is aborted before it can observe a inconsistent state).

The STO design attempts to resolve tensions between usability, generality, and performance. Memory transactions can access tens or hundreds of distinct logical items, so for any hope of good performance, the item representation must be compact, uniform, and small. For any hope of generality, however, items must support storing arbitrary data and implementing arbitrary version checks and data structure manipulations. Many aspects of our design aim for a productive middle ground between the minimality and uniformity required for performance and the usability desired for datatype designers.

Figure 4.1 shows the main STO interface; the rest of this section describes it. There are three central classes, `Transaction`, `TItem`, and `TObject`. `Transaction` represents the state of a transaction. It exposes to users interfaces for starting and committing transactions, and to datatypes it exposes interfaces for registering memory accesses. Its main state is a set of `TItems`, each corresponding to an abstract memory access. `TObject` is the base class from which all transactional objects are derived; the transaction system uses its methods to delegate parts of the commit protocol.

```
// §4.1 Methods called from user code
class Transaction:
    Transaction();      // start new transaction
    void abort();       // abort with exception
    // commit, or abort without exception
    bool try_commit();
```

```
// §4.2 Called by datatypes during txn execution
class Transaction: ...
    // return item for a logical subobject of object,
    // creating and adding a new item if necessary
    // (subid is the subobject ID):
   TItem& item<T>(TObject* object, T subid);

class TItem:                    // size 4 words (32B)
    Transaction& transaction();
    TObject* object();
    T subid<T>();               // subobject ID

    // manage read-set
    bool has_read();
    R read_version<R>();
    // add read item, prefer oldest read_version:
    void add_read<R>(R read_version);

    // manage write-set
    bool has_write();
    W write_value<W>();
    // add write item, prefer newest write_value:
    void add_write<W>(W write_value);

    // manage user flags (opaque to STO)
    unsigned flags();
    void set_flags(unsigned flags);
```

```
// §4.3 Called at commit by the transaction system
class TObject:
    // called in phase 1 to lock write items:
    void lock(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 in phase 3 to unlock write items:
    void unlock(TItem& it);
    // called after commit/abort to clean up as needed:
    void cleanup(TItem& it, bool committed);
```

```
// §4.4 Methods related to opacity checking
class Transaction: ...
    // called by install():
    version_type commit_version();
    // validate opacity, or abort with exception:
   void check_opacity(version_type version);
```

Figure 4.1: The core STO interface, organized by function.

4.1 Using transactions

Here's a transaction that increments one variable and decrements another:

```
TBox<int> x, y;              // transactional int objects
while (1) {
    Transaction t;             // open new transaction
    x.t_set(t, x.t_get(t) + 1);     // 'x = x + 1'
    y.t_set(t, y.t_get(t) - 1);     // 'y = y - 1'
    if (t.try_commit())            // try to commit
        break;
}
```

The `TBox` parameterized type (§6) wraps an object of arbitrary type—here, an `int`—in a container that interfaces with the transaction system. (`int`s don't on their own supply the locks and versions required to implement serializable transactions.) A `TBox` supports two transactional operations, `t_get` and `t_set`, that transactionally read and write the wrapped value. `TBox<T>` is a subclass of the `TObject` base class shown in Figure 4.1.

The `Transaction` type is STO's type for transactions. Declaring a `Transaction` object starts a new transaction. (Alternatively, the user can reuse an existing `Transaction` object, which offers some speed advantages; our benchmarks use one `Transaction` object per thread.) The user then passes the transaction as an argument to the appropriate transactional operations. When the transaction ends, the user calls `t.try_commit()` to execute the commit protocol. The return value for `t.try_commit()` indicates whether the commit succeeded; the code above retries the transaction until success.

In user code that manipulates larger transactions, there is often a need to initiate transaction abort-and-retry several functions away from the transaction's declaration. We implement such aborts with C++ exceptions, so a transaction is often wrapped in a `try...catch` block. Helper types and macros (and eventually language-level support) make this wrapping readable.

4.2 Transactional operations

Transactional operations are operations on concurrent data structures that are designed for use inside a transaction by an end-user programmer. Each transactional operation takes a `Transaction` argument, on which it logs read versions and written data. The names and signatures of transactional operations are datatype-specific. End-user programmers use transactional operations in much the same way they might use non-transactional operations on a data structure. The implementations of transactional operations, however, are much more intertwined with the STO system.

A transactional hash table mapping integers to strings might, for example, provide this method:

```
bool THashtable::t_insert(Transaction& t, int key, string value);
```

`THashtable` is a subclass of the `TObject` superclass representing any transactional datatype. The `t_insert` method inserts the `key, value` pair into the hash table, returning `false` if `key` is already present. This will use some form of concurrency control, such as bucket locking, to ensure correctness in the presence of other concurrent operations, and this concurrency control often resembles that required for non-transactional concurrent operation. However, the transactional method must also log logical reads and writes to the `Transaction` argument so that the insertion happens transactionally. Specifically, any inserted data must be hidden until the transaction commits, for instance by stashing the inserted data in `t`'s write set.

The transaction system's interface for transactional operations revolves around `TItem` objects. A `Transaction`'s primary state is a set of `TItems`, where each `TItem` stores the portion of the transaction's state for a single *logical subobject* of some transactional object. A logical subobject is identified by a pair of a `TObject` and a subobject ID, or "subid" (of arbitrary type, but usually an integer or pointer). The `Transaction::item(object, subid)` method returns the unique `TItem` associated with that pair (possibly after adding it).

The exact meaning of "logical subobject" depends on, and is determined by, the relevant transactional datatype. In general, a subobject corresponds to a portion of a data structure whose updates don't conflict with other subobjects. A transactional array, for instance, might have one subobject per array element, with another subobject for the array's size. A transactional hash table might have one subobject per element and one subobject per bucket (for validating `t_find` operations that return "key not found"). A transactional box has exactly one logical subobject, since all writes

```
class Array<T> {
    T* a_;
    int n_;
    lock_type* locks_;

    T get(int i) {
        locks_[i].lock();
        T value = a_[i];
        locks_[i].unlock();
        return value;
    }
    void set(int i, T value) {
        locks_[i].lock();
        a_[i] = value;
        locks_[i].unlock();
    }
}
```

Figure 4.2: A simple concurrent array. Every entry has its own lock. We use an exclusive lock for simplicity; performance-oriented code would likely use read/write locking or version validation to avoid contention on reads. Initialization code and bounds checking not shown.

```
class TArray<T> {
   T* a_;
   int n_;
   version_type* vs_;

   // called by users:
   T t_get(Transaction& t, int i) {
      TItem& it = t.item(this, i);
      if (it.has_write())
         return it.write_value<T>();
      else {
         vs_[i].lock();
         it.add_read(vs_[i]);
         T value = a_[i];
         vs_[i].unlock();
         return value;
      }
   }
   void t_set(Transaction& t, int i, T value) {
      t.item(this, i).add_write(value);
   }

   // called by transaction system:
   void lock(TransItem& it) {
      vs_[it.subid<int>()].lock();
   }
   bool check(TransItem& it) {
      int i = it.subid<int>();
      version_type v =
          it.read_version<version_type>();
      return vs_[i].same_version(v) &&
          (!vs_[i].locked() || it.has_write());
   }
   void install(TransItem& it) {
      a_[it.subid<int>()] = it.write_value<T>();
      vs_[it.subid<int>()].increment_version();
   }
   void unlock(TransItem& it) {
      vs_[it.subid<int>()].unlock();
   }
}
```

Figure 4.3: A transactional version of the array from Figure 4.2.

on the contained element conflict with all reads of that element.

A `TItem`, then, comprises four pieces of state: a transactional object, a subid, an optional *read version*, and an optional *write value*. The object and subid are set when the item is created and cannot be changed. The read version and write value are set in a transactional operation to indicate that the logical subobject has been read at a particular version or written with a particular value. The read version and write value can have arbitrary type, although the read version is typically

a 64-bit version number. The presence of a read version indicates to the transaction system that a logical subobject should be verified during the commit protocol. The presence of a write value indicates that a logical subobject should be locked during the commit protocol, and that its `install` method should be called if the transaction commits.

Consider, for example, an array. Figure 4.2 shows a simple concurrent array with one lock per item; the `get` method returns the element at a given index, and the `set` method sets the element at a given index. Figure 4.3 shows a transactional version of the same array. Some important features are as follows; we have found these features to be shared across many of our data structures.

—The `t_set` method simply adds a write value to the transaction. The same index might be written multiple times; since later writes should override earlier ones, the value stored in the `TItem` is that from the most recent `add_write`.

—The value stored by `t_set` is installed via a sequence of `lock`, `install`, and `unlock` calls initiated during the commit protocol. The combination of these calls is quite close to the functionality of the original `Array::set` method. The only difference is that the one-bit locks in `Array` have been expanded in `TArray` to 64-bit version numbers, which allow the detection of concurrent modifications.

—The `t_get` method is more complicated than the original, since it needs to check whether the index was previously written in the same transaction. If so, the previous write value is returned. Otherwise, the method reads the value as in the original `Array::get`, and additionally adds the read version to the `TItem`. A transaction might read an index multiple times and, in case of concurrent modification, the second read could return a different version. To ensure that such a transaction aborts, the transaction system stores the *first* read version encountered per subobject: later read versions are ignored (though if the read version does change, this will be caught by an opacity check if enabled).

—Intuitively, the `check` method verifies that a read subobject has not changed since the time it was originally read. This is usually just a version check (implemented, of course, in a type-specific way). However, `check` must fail if the relevant subobject is currently being modified by a running transaction—unless, of course, that transaction is the present transaction (a subobject might be both read and written). The three-part check shown (same version, and not locked unless `item.has_write()`) is typical.

We also found it useful to provide a few flags on each `TItem`; for instance, a flag might indicate that a subobject has been deleted. The transaction system doesn't interpret these flags. When first created, a `TItem` has no read version or write value and its flags are zero.

`TItem`'s read version, write value, and subid have arbitrary type. Trivial objects, such as pointers and integers, are stored directly in the `TItem`. More complex objects, such as strings, are stored elsewhere; the transaction system takes ownership of such complex objects and ensures they are destroyed when the transaction completes. The author of a transactional object is responsible for ensuring that the type used to extract a read version, write value, or subid corresponds to the type used when adding it. It is an error, for example, to add an integer write value and then extract it using a string type. We'd like to add a debugging mode to catch such errors.

The efficiency of `TItem` operations was a major performance concern. It was particularly important that `TItems` be small, fixed-size objects; in our implementation a `TItem` takes four words (32 bytes). Several features of the design arise from this requirement. For example, in an initial design, `TItem` was actually a base class; each transactional datatype created appropriate derived classes that implemented the behavior necessary for the commit protocol. This design was in some ways simpler and less error-prone—no concerns about extracting a write value using the wrong type, for example—but it required one expensive allocation per `TItem` and was quickly abandoned.

4.3 Commit protocol

STO uses optimistic concurrency control [27] for its commit protocol, as described in §1.1. While an untyped STM manipulates memory words and versions directly, STO maps item-specific parts of the commit protocol to methods on the `TObject` base class. Each subclass overrides these methods, providing implementations specialized for the relevant data structure.

`Transaction::try_commit` executes the commit protocol's phases. In Phase 1, the write-set (the set of `TItems` with a write) is sorted according to a globally consistent order (we use the `TObject` and subid to obtain this ordering) and then the `lock` method is called on each item's `TObject`. In Phase 2, the read-set (the set of `TItems` with a read) is verified by calling the `check` method of each item's `TObject`. If any call returns false, the transaction aborts. In Phase 3, the write-set is installed with `install` method calls, and after the transaction ends (whether successful or not) the write-set

is unlocked with `unlock` method calls. Finally, the transaction system calls a `cleanup` method for each item to execute any necessary cleanup tasks; for example, if a transaction that inserted a new item ends up aborting, the new item's cleanup task would actually remove the allocated item from the data structure.

4.4 Opacity

In a transactional system with *opacity*, no transaction can observe an inconsistent state: if a transaction reads two values `v1` and `v2`, then there must have existed a state in the serial transaction order when those values had those versions simultaneously. This doesn't mean that the transaction will eventually commit; for instance, `v1` might have changed by the time `v2` was read. But opacity is desirable on its own. In a system without opacity, a transaction can observe an inconsistent state. Although such a transaction cannot possibly commit, the inconsistency might in the meantime cause assertion failures, infinite loops, and programmer confusion.

Unfortunately, opacity can also be expensive. The obvious opacity check, which we call *direct* opacity, simply revalidates every past read item on each new read (and aborts on failed validation). This has potentially disastrous $O(n^2)$ overhead (n is the size of the transaction).

A better alternative was introduced by TL2 [12]. This algorithm requires that all read versions use a common version number space. When a transaction begins, the transaction system takes a snapshot of a current global version number; this snapshot is called the *version bound*. When an item is read at version v, the system compares v with the bound. If v is no larger than the bound and the subobject is not locked, then observing v cannot cause an opacity violation. If the subobject is locked or v is larger than the bound, however, then reading v might cause an inconsistency; in TL2, such a read causes the transaction to abort. When the transaction commits, the transaction system obtains a *commit version* by atomically incrementing the global version number. The transaction's written subobjects are marked with this version.

TL2 opacity is usually faster than direct opacity, but it too has performance problems. Each transaction updates a shared variable, which can cripple scalability at high core counts. (Even the TL2 "low contention global version clock" [12] has scalability problems.) Since some applications, such as our database, do not require opacity, the STO system does not force datatypes to provide

it, and a non-opaque transaction will incur no opacity overheads.

A transactional object providing opacity should use TL2-style version numbers; the `Transaction::commit_version` method, which should only be called from `install`, provides access to the commit version for the current transaction. When making a read, the object calls `Transaction::check_opacity(v)`, passing in the read version `v`. This will perform an extended TL2 version check. The initial check follows TL2, but a failure does not immediately cause an abort. STO instead reloads the version bound and performs a full read validation, aborting only if that fails. This precise validation (which is the same check that would happen at commit time) will succeed if and only if all reads so far are consistent with one another. This extended check improved performance on our benchmarks.

4.5 Discussion

The requirements for correct transactional operation under STO (locking, version numbers) are quite close to the requirements for correct concurrent operation even without transactions. For instance, adding STO support to a fast concurrent B-tree variant [29] required few data structure or code changes. We thus believe that many data structures can support both transactional operation and non-transactional operation concurrently, with significant code overlap between these modes. It does, however, require thought to ensure that non-transactional updates to a data structure don't violate transactional correctness. The main key is to ensure that the `check` operation, and (when appropriate) the TL2 opacity check, will notice any non-transactional changes that occur.

Although STO's central commit protocol resembles optimistic concurrency control, this choice is not forced on datatypes. A datatype could instead, for example, acquire locks at execution time, using the `cleanup` method to perform any necessary undo operations in the event of an abort (though the datatype would still have to handle the usual problems with pessimistic methods like deadlock).

There is some middle ground between using `TBox` types and implementing complex data structures. Consider as an example a counter, where we only care whether or not the counter has value zero. A `TBox` around this counter would abort any time the counter changed, even if it remained above zero. However, it is not difficult for an application developer to implement a simple `TObject` around this counter, which implements for `check` a confirmation that the counter's value remains

positive. This idea is further explored in §7.4.

STO currently acquires locks on logical subobjects in a global order based on object pointer and subid. This effectively enforces a global lock order on the rest of the system. A better choice might be for STO to lock with bounded spinning.

Chapter 5

Implementation

Our implementation of STO is 6,886 lines of C++ code, with 1,038 in the core transactional library, 1,861 in tests, and the rest in various transactional data structures. The rest of this section describes important implementation features and optimizations.

Choice of subid Choosing the proper subid for a given subobject is often an important performance choice. Consider, for example, an update to an existing key in a hash table. One obvious choice to identify the subobject of this update would be the string key that corresponds to this element in the table. This choice of subid, however, then requires copying a (potentially very large) string key for every hash table update, and performing string comparisons to search for the unique `TItem`. This also presents a challenge for ordering items to prevent deadlock (the same string key can be stored at different pointer locations, for instance, so it is unsafe to order by subid pointer value). In this case the best solution might be for a datatype that uses string subids to implement bounded spinning (aborting after a certain period of time waiting for a lock), allowing the rare deadlock to be resolved by aborts.

These performance concerns and complications can be avoided by instead choosing an integer or unique pointer subid. Integers and pointers can be stored directly in a `TItem`, require no allocations or copies, and also make comparisons very cheap. Choosing a proper pointer for a given logical subobject can be challenging; importantly, the pointer must have the property of being unique for any operations—including both reads and writes—of the subobject. For the hash table example above, a good choice of subid is the pointer to the node in the hash table representing the given element. This pointer also provides instant access to the node at commit time, avoiding the

second lookup into the hash table that a string key would require. For unsuccessful get operations, there is no such pointer to use, so we instead use the index of the bucket that the key hashes to.

`TItem` uniqueness. STO requires that each logical subobject in a `Transaction` correspond to a unique `TItem`. Our initial implementation ensured uniqueness with a linear item search, leading to $O(n^2)$ item comparisons (n is the transaction's size). We then applied some optimizations from untyped STMs, such as allowing the `Transaction` to store duplicate read items in certain situations [12]. Our implementation allows a data structure to request read-only items: these items will not search the transaction set if no writes have yet been performed, since in that case there are no prior write values we might need to observe for transactional correctness. Once a write has been made, read-only items revert to the standard behavior of searching the transaction set (starting at the index of the first item written). This optimization can make a significant performance improvement in some applications, particularly those that have large read-only transactions. However, because it also complicates the model for datatype designers (`TItems` are no longer always unique), this behavior must be explicitly requested by a datatype, so datatype designers do not have to consider this more complicated model unless needed for performance. Importantly, although read-only items can complicate the implementation of a datatype, a datatype that uses read-only items becomes no more difficult for an end-user programmer to use.

Still better performance on large transactions was obtained by adding a simple hash table for $O(1)$ item lookup, though for small transactions this did not help and is not used. The hash table maps from a hash of an item to its index in the transaction set, and conflicts favor the most recently added item. The hash table has the further advantage of improving performance without adding any complication to our model, and thus is always enabled for large transactions. The performance impact of these techniques is investigated further in §7.2 and §7.5.1.

`TItem` modifications. Some datatypes occasionally modify or erase previous read- and write-set items. For instance, consider a B+ tree transaction that first scans a key range, then inserts a key into that range. The natural implementation of range scan treats the scanned leaves as logical subobjects and adds their node-level versions to the read-set. But the insertion will change these versions (to invalidate range scans in other transactions). Of several designs we tried, the most natural was to update the relevant read versions when this occurs, which requires special `TItem` interfaces. We are careful to ensure transactional correctness by verifying the old read-set version

first.

Garbage collection. Many fast concurrent datatypes in C use Read-Copy-Update (RCU) [14, 30] to free memory. RCU effectively acts like a garbage collector; when it is present, datatypes can use advanced forms of concurrency control that avoid read locking. Any such garbage collector requires access to roots, and in STO, transactions store such roots. We therefore provide an epoch-based reclamation RCU implementation [18] (and allow datatype designers to integrate their own). An STO datatype can register objects to be freed, and callbacks to be called after any concurrent transactions are guaranteed to have finished. Epoch-based reclamation works as follows: we periodically increment a global epoch value. Each thread then stores the global epoch it observed at the start of its current transaction. For a pointer registered at epoch g, we know there can be no threads that still have access to this pointer if every thread in the system has a local epoch of at least $g+1$, at which point it is safe to free the pointer.

Chapter 6

Datatypes

This section describes interesting features of some of STO's transactional datatypes.

`TBox<T>`. The box type is our most basic data structure, wrapping an arbitrary type for use in our transactional system. This only requires adding a version number and implementing the `TObject` methods. A `TBox` supports two operations, `t_get` and `t_set`.

`TArray<T>`. Arrays offer indexed get and set methods. An array can be written as a C array of `TBox`es, or can be custom written to do the work of a `TBox` for each of its elements similar to that shown in Figure 4.3. The latter requires only one `TObject` and corresponding virtual method table, whereas the former has one `TObject` per element in the array, but this does not make a significant performance difference. An important departure from Figure 4.3 in our implementation, however, is that `t_get` does not acquire any locks, which both improves the speed of get operations and avoids any reader-reader contention. Instead, we check that the version number is not locked, read the desired value, and then make sure that the version has neither changed nor been locked since our previous read, retrying if it changed. Because any write to the array must acquire the lock and then increment the version number, we are guaranteed to have a consistent read of the array value if the version number remains constant and unlocked.

Our arrays, and in fact all of our containers, can contain values of arbitrary type (including other `TObject`s). This requires care both for correctness and optimal performance. An array of strings or other complex values, for instance, must make copies of these values, while an array of integers need not perform any copies. The transaction system already handles its part automatically: when setting the write value of a `TItem`, the transaction system will store the value directly in the `TItem`

if it is a simple type fitting into one memory word, and will otherwise allocate the necessary space, copy the value, and clean everything up after the transaction ends. This is accomplished transparently to the datatype programmer using C++ templates. Our generic datatypes add similar support for arbitrary values stored in the data structure, using C++ templates to determine whether or not a particular choice of value requires copying. More than that, however, while our locking schemes ensure that no two threads ever write to a particular value at the same time, a reader may observe a value as it is being updated. For some choices of value—including the C++ standard containers—simultaneous reads and writes are not safe, and can even lead to segmentation faults (if a reader observes the old value as it is freed, for instance). Thus, for generic, non-trivial values we must perform updates atomically. Like many problems in computer science, this requires an extra level of indirection—we store in our data structure a pointer to a particular value type (only when it is a non-trivial type), and updates then allocate a new pointer to a new value, updating the data structure's pointer in a single instruction (and using RCU to free the old memory).

`TList<T>`. This is a typed singly-linked list with a separate size component; its main operations are insertion, deletion, lookup, and iteration. Storing a separate size component transactionally can be somewhat difficult. Transactional reads of the size must consider any earlier inserts or deletes in the current transaction. For this we add an item to the transaction set that tracks only the current size offset of the transaction (and does nothing at commit time). Additionally, a separate size variable requires an insert/delete operation to happen atomically with the size variable update. This requires any transaction that inserts or deletes to lock the whole list, reducing concurrency. Still, for workloads (such as the STAMP benchmarks) that require frequent polling of the list size, a size variable is important. Our list also supports transactional iteration. Since looking up a specific item can be accomplished with the lookup operation, we make the assumption that iteration will generally be used to traverse the whole list. Then, rather than tracking every node traversed, an iteration tracks a version for the whole list, which is invalidated by any insert or delete operation (this is also how the transactional size operation must be implemented). While this can actually *increase* false conflicts of our list relative to an untyped implementation, for the very common case that the whole list or at least a large portion of it is traversed, we store many fewer words, improving performance. (This is especially true for lists containing immutable values—a property that is often true in STAMP's benchmarks—since any reads of immutable values during iteration

need not add anything to the read-set.)

`TListSet<T>`. This extension of `TList` implements a set (without duplicates); the type `T` must support equality, and the transactional `t_insert` method fails if the element is already present. Thanks to the placeholder-item technique described in §3.1, transactional insertions of different items avoid all conflicts.

`TQueue<T>`. Our transactional queue supports two operations, `t_push` (add an element to the tail) and `t_pop` (remove an element from the head). We observed that a `t_push` does not actually observe the queue's state, so STO's `t_push` implementation doesn't modify the read-set, and `t_push` will never cause a transaction to abort. In an untyped STM, in contrast, *any* two `t_push` operations conflict (because a push must read the queue's state before writing to it).

`t_pops` always conflict with each other, while `t_push` and `t_pop` only conflict in the case of an empty queue. This is a slightly tricky case, which we handle as follows: a `t_pop` that observes a non-empty queue succeeds so long as no other pops have since occurred on the queue at commit time (this is tracked using a "pop" version number, which is incremented after every pop). A `t_pop` that does observe an empty queue succeeds only if the queue remains empty at commit time. It thus logs the current "push" version number (incremented after every push operation), and the commit succeeds so long as no new pushes have occurred. Pops of a globally empty queue also check their write-set for pushes in the current transaction and pop from those, if applicable.

`THashtable<K, V>`. Our hash table supports `t_get`, `t_insert`, `t_put`, and `t_delete` operations. Each element is a logical subobject, as is each hash bucket. Transactional inserts again use placeholder items. Inserts modify both the item's version number and the version number on the corresponding bucket. When an item is found, the `t_get` method adds the found item to the read-set; when not found, it adds the containing bucket, ensuring that a later insert will abort the transaction.

Deletes mark a node invalid at commit time (forcing any observers of this node to abort), then remove it from the hash table and schedule it for garbage collection with the RCU system. Note that deletes need not modify the bucket version, as the bucket version is only used to invalidate unsuccessful get operations; a delete never changes the status of an unsuccessful get. Our hash table does not currently support iteration. However, we would most likely add it in a similar manner to list iteration, where we log the bucket versions for each bucket traversed (this would also require making deletes increment the bucket version number).

`TMasstree`. Masstree [29] is a high-performance concurrent B+ tree-like 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 logical subobjects correspond to the values stored in the tree and, to support range queries and unsuccessful gets, tree leaf nodes (which will change following an insertion or deletion). Only very small changes were required in Masstree to support transactional operation; specifically, our Masstree exposes the previous version numbers of leaf nodes split as a result of an insertion, which `TMasstree` uses to update the read-set as described in §5.

An important optimization was storing the string values of a `TMasstree` in the same allocation as the associated version number (avoiding extraneous allocations and cache line accesses), as well as updating the value in-place. Since we store strings in the tree as simple byte arrays, it is safe to update the value in-place (a reader that observes a value as it is updated will never traverse beyond the bounds of the allocation, and will notice immediately afterwards that the version number either changed or is locked, and retry). We cannot update the string in-place if the new string is larger than the capacity of the current string value, however. In this case we must do a locked update through Masstree to update the value to a new pointer. In order to avoid performing this update at commit time (which would require storing the string key for this element), we decide whether we need a capacity change *at execution time*. If so, we do the locked update then, resizing the capacity of the string value while keeping the contents the same. Then at commit time we can perform an in-place update, which is now guaranteed to have the required capacity to succeed. Like our other datatypes, `TMasstree` supports generic values, so for non-string values it acts the same as other datatypes and decides whether it can do direct updates or if it needs an extra level of indirection (when the value type is non-trivial).

`TUntyped`. Finally, the `TUntyped` type implements an untyped STM: its `install` method makes changes to arbitrary words of memory. `TUntyped` both shows the generality of our system and is critical for supporting the STAMP benchmarks. It stores version numbers by hashing memory addresses into a fixed-size array, while associating write values directly with destination addresses. We were able to use our `TArray` implementation directly for this—when we write to the array, the array will then lock that entry during the lock phase, locking out any other writers from that word address. At install time, the array will increase the version number of the given entry, forcing any

reads to that entry of the array to abort. Since multiple words could hash to the same entry in the array, there are potential false conflicts. As long as we use a sufficiently large array size though, this should not be a significant problem. Because we can use our existing array implementation so effectively, `TUntyped` only requires 40 lines of additional code.

Chapter 7

Evaluation

This chapter evaluates STO as follows:

- Microbenchmarks that demonstrate STO's scalability with respect to number of cores, the overheads of its opacity, and the effects of our hash table optimization (§7.2).
- A comparison of STO to untyped STMs using STAMP, demonstrating the benefit of typed transactions (§7.3).
- An evaluation of the effectiveness of application-defined objects in STO using the vacation benchmark of STAMP (§7.4).
- A comparison with Silo, a fast main-memory database, where we show that STO can outperform a purpose-built transaction system (§7.5).
- An evaluation of STO's ease of use for end-user programmers, by applying STO to a previously sequential network server (§7.6).

7.1 Experimental setup

All our experiments were run on a single machine with 2 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.

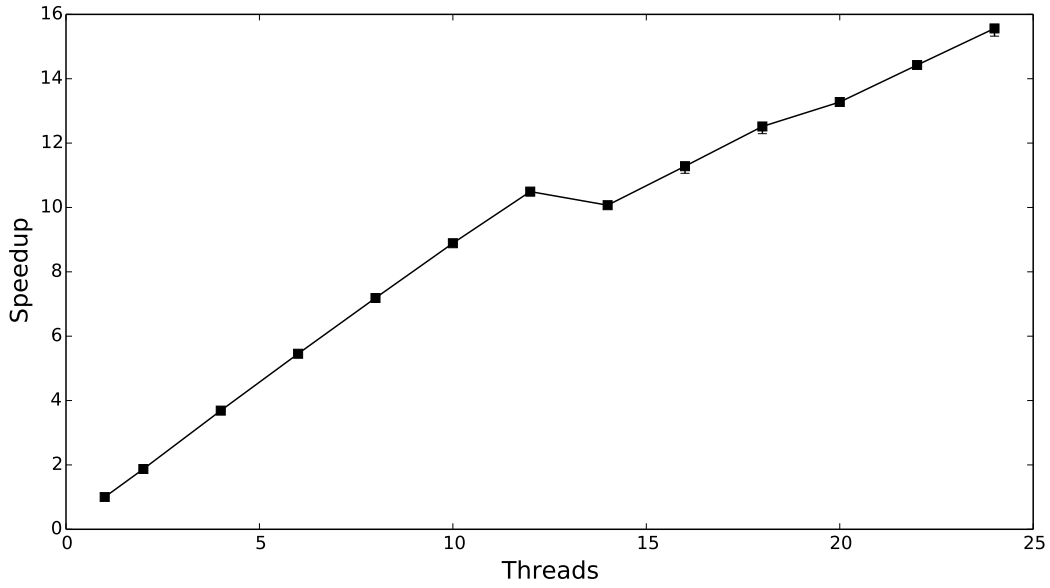


Figure 7.1: STO's scalability, with 10-item transactions.

In all graphs, we report the median of 5 consecutive runs, with the minimum and maximum shown as error bars.

7.2 Microbenchmarks

We first run a series of microbenchmarks to evaluate the inherent overhead of STO's generic transaction system. Each experiment performs transactions that read or increment random keys in a transactional array of integers. Transaction size varies, but each transaction consists of half reads and half increments. This data structure is so close to untyped memory that STO offers few benefits over untyped STM. We ask several questions: Does STO have any inherent overhead that would limit scalability? What are the performance consequences of extended TL2 opacity? How does our hash table optimization scale with large transactions?

Scalability without opacity. Figure 7.1 shows the scalability of STO with 10-item transactions under low contention (array size 1 million, $< 1\%$ abort rate at 24 cores) and without opacity. Scalability is good, though not perfectly linear at larger core counts. Much of this drop is due to memory accesses that cross the 12-core socket boundary. Although scalability is clearly limited under high contention, we still aim for STO to achieve some amount of scalability under con-

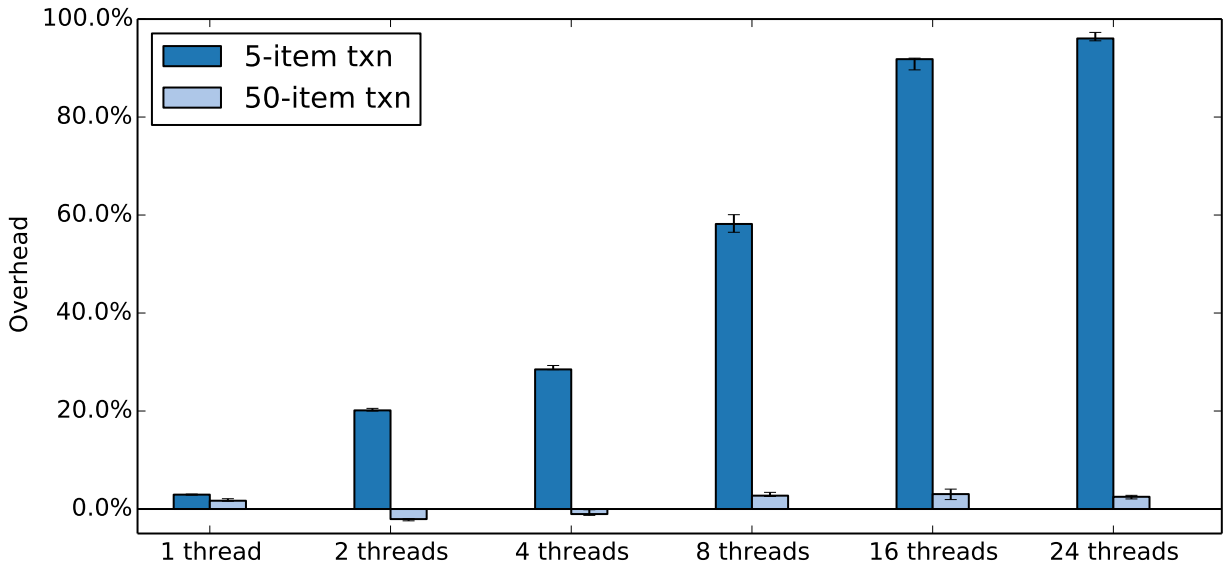
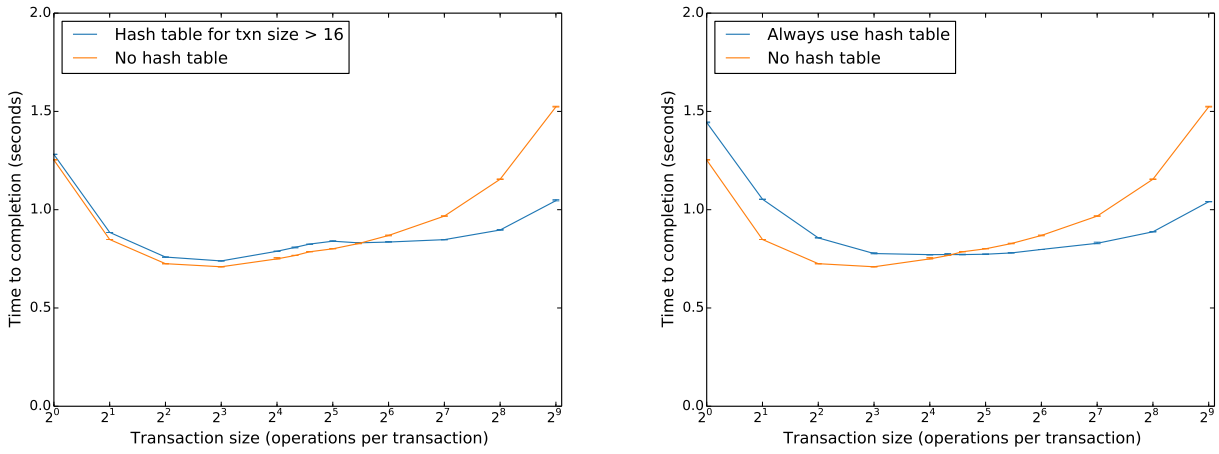


Figure 7.2: The overhead of TL2-style opacity.

tentious workloads. On a workload consisting of 100-item transactions and a 10,000 element array (two transactions have about a 40% chance of conflicting), STO without opacity still had a 2.3x speedup at 24 cores (with an abort rate of 45%) compared to a single core.

Opacity. Extended TL2 opacity affects scalability since transactions atomically increment a shared global variable to obtain their commit versions. The associated cache line bouncing could throttle performance, particularly for small, cheap transactions; we would expect large transactions, or transactions whose `install` methods are expensive, to contain enough computation to mask this effect. We measured the overhead of extended TL2 opacity relative to no opacity under low contention (array size 1 million) for both small (5-item) and medium (50-item) transactions. Figure 7.2 shows the results. As expected, opacity is quite expensive for small transactions, costing 20% even when there are only 2 cores, and nearly halving transaction throughput at 24 cores. For medium-sized transactions, the story is different. Opacity never has terrible overhead, and actually speeds up execution at small core counts (2 and 4) by aborting early and avoiding wasted work. We also evaluated direct opacity, in which every opacity check corresponds to a full read validation. This avoids the scalability bottleneck of the global version number, but suffers much higher overhead. As expected, overhead grows with transaction size but remains independent of core count; it was roughly 10% for 5-item transactions and 50% for 50-item transactions at all core counts. This suggests that a transaction system built for short transactions should perhaps implement direct



(a) (b)

Figure 7.3: (a) Scalability with respect to transaction size with and without our hash table (number of transactional operations is kept constant throughout). Lower numbers are better. In (b) we enable our hash table even for small transaction sizes. This confirms our choice of 16 as the approximately optimal cutoff for when to begin using the hash table.

opacity.

Hash table optimization. As explained in §5, we were able to improve performance for large transactions significantly by using a hash table for lookups in our transaction set. The exact effects of this hash table can be seen in Figure 7.3a, which shows the overhead of increasing transaction size with and without our hash table (we hold the total number of operations in this experiment constant—so the ideal graph would be a flat line). Lower numbers are better. Note that the x-axis is a log-scale. Although we never use the hash table for small transactions, there is a small, constant overhead for transactions of size less than about 40—likely from the additional branch and the increased size of the code. Above that, however, the benefits of the hash table are clear, as we scale near linearly even for larger transactions. The hash table used in this benchmark has a fixed size of 512, so we can see that its effectiveness starts to drop off as the transaction size approaches 512 and hashing conflicts become much higher.

We chose 16 as the cutoff point for when to start using the hash table. In Figure 7.3b, we compare always using the hash table (even for small transaction sizes) to no hash table. This helps confirm our choice of 16, as this is near the point where always using the hash table provides an improvement over never using a hash table. Although Figure 7.3b suggests an exact value of about

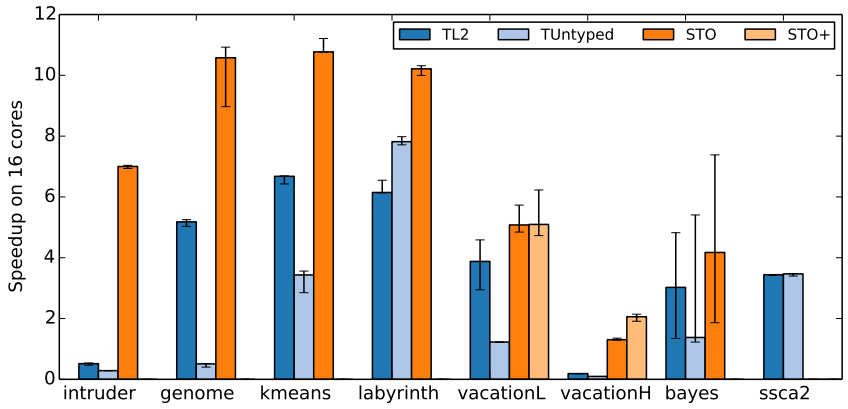


Figure 7.4: STAMP results on 16 cores. Larger bars are better; there is no STO result for ssca2 because that benchmark uses no datatypes.

Benchmark	**Arguments**
intruder	-a10 -l2048 -n10000 -s1
genome	-g16384 -s64 -n16777217
kmeans	-m160 -n160 -t0.01 -i inputs/random-n262144-d32-c16.txt
labyrinth	-i random-x512-y512-z7-n512.txt
vacationL	-n2 -q90 -u98 -r1048576 -t4194304
vacationH	-n8 -q1 -u60 -r1048576 -t4194304
bayes	-v32 -r4096 -n10 -p40 -i2 -e9 -s1
ssca2	-s21 -i1.0 -u1.0 -l3 -p3

Table 7.1: Configurations used for the STAMP benchmarks

20 for when the hash table becomes optimal, with a cutoff value of 16 the actual crossover point is somewhat higher, at around 40 (because we cannot predict transaction size, and so do not start using the table until the transaction size is 16). Since the overhead of the hash table when used with a cutoff of 16 is never particularly large, and it is most useful for transactions of size at least 100, we consider this tradeoff acceptable.

7.3 Typed transactions: STAMP

We now test our hypothesis that STO's typed transactions can outperform untyped transactions, thanks to smaller read- and write-sets and fewer false conflicts. To do so, we ported the STAMP transactional-memory benchmark suite [31] to STO. STAMP is a widely-used standard for evalu-

ating hardware and software transactional memories; it was designed for an untyped STM, and ships with a variant of TL2. It comprises eight benchmarks that model different parallel coding patterns. Although the benchmarks were written for an untyped STM, they use several data structures—lists, queues, maps—that make the appropriate untyped STM accesses. We first ported seven of the STAMP benchmarks[1] to our "untyped" STM, the `TUntyped` object; and then designed interfaces to our transactional data structures that match STAMP's interface. We can thus measure three configurations—the original STAMP benchmarks with TL2; STAMP with STO `TUntyped`; and STAMP with STO data structures—and address several questions. First, how does STO compare to a tuned untyped STM (TL2)? And second, is the performance difference due to typed transactions, or to some other aspect of the STO implementation?

We make several changes to the STAMP benchmarks in all configurations. First, we implement maps using hash tables rather than balanced trees. This actually favors untyped transactions, since compared to tree operations, hash table operations make many fewer accesses to shared memory (and avoiding such accesses is a primary STO goal). Second, some of STAMP's suggested parameters produce datasets too small to scale to high core counts. For instance, given the suggested intruder parameters, no STM ever beat sequential performance: contention was simply too high. We increased the data size for some benchmarks to mitigate these scalability problems and ensured that TL2 performance was still comparable to that in the STAMP paper. The configurations used are shown in Table 7.1.

Figure 7.4 shows the results at 16 cores (results at 4 cores were similar, except that the bayes benchmark was more variable; the STO+ results for vacation are explained in the next section). For each benchmark, we graph the speedup for each STM implementation over STAMP's sequential code for that benchmark. Despite the wide range of workloads, contention levels, and data structures represented in STAMP, STO always beats TL2, often substantially. Only STO comes anywhere close to scaling up to 16 cores. Furthermore, untyped STO usually performs worse than TL2. While this could be in part because our untyped implementation is not as fine-tuned as TL2's, it does suggest that STO's benefits come from the use of datatypes, rather than (for instance) a better core transactional system.

STO performs best on the genome and intruder benchmarks. These heavily use lists, which,

[1] Yada was eliminated because of bugs in the benchmark. This seems to be a common issue [19].

in untyped STMs, add many extraneous words to read- and write-sets (i.e., one for every "next" pointer encountered). An intruder transaction using `TUntyped` contains, on average, 20x more items than the same transaction using full STO, and the maximum ratio is much worse (200x). The results for genome are less extreme, but still significant, with 3x more items in `TUntyped` on average, and 20x more items in the worst case. `TUntyped` is generally slower than TL2, sometimes significantly so. This is likely in part because our implementation is less fine-tuned than TL2, and also because of STO's generality overheads. (For instance, STO items are larger than TL2's, and because TL2 understands the semantics of its items, it need not always ensure unique storage of an item in its read- and write-sets.) However, on labyrinth, `TUntyped` outperforms TL2. The reason is our extended opacity: TL2 has 26% aborts, while our additional read-set validation reduces the abort rate to 18%. (Most of the other benchmarks have lower abort rates, so extended opacity checking does not come into play.)

7.4 Application-defined operations

Users can make further gains over STO by implementing their own transactional objects. We demonstrate this with the vacation benchmark, which models a reservation system. When two clients in the benchmark happen to make a reservation for the same item (such as a hotel room), they will conflict, even though semantically both reservations can succeed so long as there are still equivalent items available to reserve. To solve this problem, we implemented a custom transactional object for reservation items. This object tracks the number of equivalent reservable items, and its `check` method for reservations reports a conflict only if the price of an item changes or there are no more items to reserve. This system is referred to as STO+ in our graphs. Under low contention ("vacationL"), STO+ performs equivalently to STO (and with roughly 25% better speedup than TL2). Under higher contention ("vacationH"), however, STO+'s application-specific conflict detection means transactions observe 5x fewer aborts than STO (1% abort rate, rather than 5%), and the benchmark as a whole has about 1.6x higher speedup.

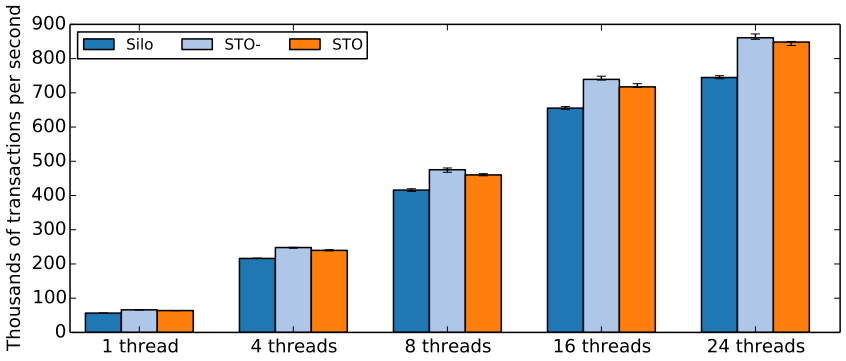


Figure 7.5: Results for TPC-C (standard mix) for Silo, an in-memory database built using ad-hoc concurrency, and STO, which is a simple combination of STO's `TMasstree` and its `Transactions`. In "STO-", we disable opacity and read-my-writes (Silo also disables these features). Silo is very fast, but our version outperforms it by almost 20%.

7.5 Silo

In this section we show that despite their generality, typed transactions can outperform purpose-built transaction implementations. Our comparison system is Silo, a fast in-memory database [34]. (Specifically, we compare against MemSilo, with MemSilo's snapshots disabled.)

Silo implements transactions using Masstree; it adds to Masstree 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 [33]. The speed of Silo transactions inspired our work on typed transactions: Silo is several times faster than many other in-memory databases [34]. Our typed transaction system outperforms Silo on TPC-C by 10%, as seen in Figure 7.5[2] . Moreover, Silo's transaction system disables opacity and does not properly support reading an item that was written by the same transaction ("read-my-writes"); if we modify STO in a similar way, we can outperform Silo by almost 20% (STO- in Figure 7.5). Despite its improved performance, STO-based Silo is both simpler and more general than the original. Our transactional system plus `TMasstree` is less than 2,000 lines of C++ code. The corresponding part of Silo, however, is more than 7,000 lines long. Furthermore, Silo's implementation is deeply interlinked with Masstree, whereas adding hash table support to STO-based Silo transactions would

[2]Though TPC-C performance is often measured as New Order transactions per minute, we follow Silo in reporting total transactions per second for the standard mix.

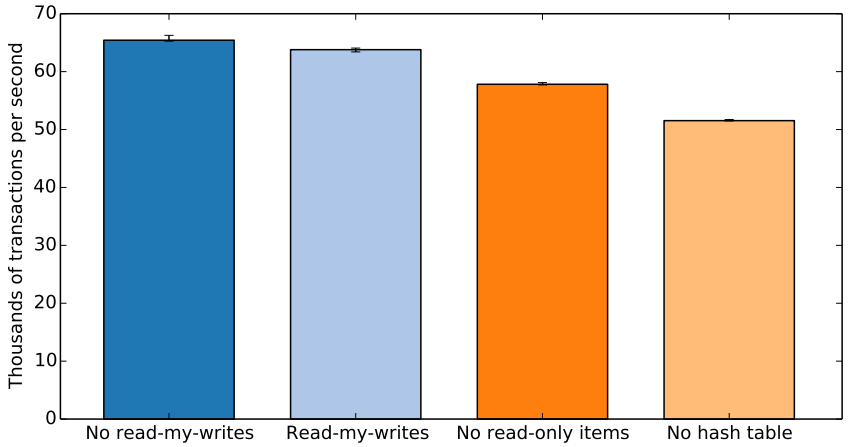


Figure 7.6: A comparison of different techniques for making reads observe prior writes in a transaction ("read-my-writes") on TPC-C. Changes are cumulative from left-to-right.

be a matter of calling methods on `THashtable` instead.

Our system has many subtle design differences from Silo (for instance, we abort if we ever see a locked version number, while Silo spins until it is unlocked), and we were unable to determine the precise reasons for our performance difference. Given enough effort, a purpose-built system for Silo transactions could match the performance of a general system like ours (just as purpose-built TL2 beats our general untyped STM). Our results show, however, that this effort may not be worth it. The generality of typed transactions has low cost, and further improvements on STO's transactional core would benefit not only STO-Silo, but also other benchmarks.

7.5.1 Item uniqueness

The TPC-C benchmark also provides a good opportunity to investigate the effects of our requirement of `TItem` uniqueness. As was shown in Figure 7.5, there is only a small performance decrease in STO when we support reading our writes and opacity. This is largely because of optimizations in our system to improve read-my-writes performance.

In Figure 7.6 we compare the effects of these optimizations when running TPC-C on a single-thread (changes are cumulative from left-to-right; opacity is enabled in every configuration). Though enabling read-my-writes performs only marginally (less than 3%) worse, disabling read-only items (which allows duplicate read items in some cases, per §5) makes this difference much more significant. TPC-C includes large, read-only transactions, which is where this optimization proves

most effective. The hash table optimization described in §5 also provides a large performance improvement in TPC-C. Together, these two optimizations provide nearly a 25% improvement for read-my-writes, allowing us to fully support read-my-writes with only minimal overhead.

7.6 Server

Finally, we briefly explore STO's ease of programming. To investigate the difficulty of applying STO-style transactions, we took a single-threaded program and made it multithreaded using STO transactions.

thttpd is a small event-driven web server [1]. Though such servers are fast enough for many web workloads, using multiple threads can matter for certain workloads, such as when connections must perform heavy computation or heavy disk I/O [26] (often multiple threads handle blocking disk I/O more effectively than available asynchronous interfaces). We converted thttpd's main event loop to use multiple threads and identified the shared read/write data structures that required concurrency control. Two core data structures required sharing between threads, a *throttling table* used to track and throttle the bandwidth of particular URL patterns, and a *mapping cache* of memory-mapped files. The throttling table corresponded to a transactional array, and the mapping cache to a hash table (for the mapping table, we ensure, using a `TBox` variant, that newly mapped files are closed if a transaction aborts). Having chosen the data structures, we simply introduced transactions around their uses. Though some uses would have been simple to implement with locks, others operated on multiple subobjects concurrently (e.g., updated multiple URL patterns atomically), and there transactions were helpful. The final code shrank by 41 lines relative to the single-threaded version.

Since the two data structures in the server that required shared access were both common data structures, we were able to use our existing transactional datatypes directly. Compared to a traditional STM, this was no more difficult to do but provided an extra speed increase. Had there been more complicated data structures, we could have just used untyped transactions to protect them, with similar results to a traditional STM.

Our conclusion, perhaps unsurprisingly, is that transactional data structures are not always the most difficult aspect of parallelizing serial code. Their availability does help the programmer introduce fine-grained concurrency quickly, however, and our transactions are not significantly

more difficult to use than those in traditional untyped STMs.

Chapter 8

Future Work and Conclusions

8.1 Future Work

There are several interesting directions for future research. One direction is modifying our approach to work with a compacting garbage-collector that relocates live objects. Another is integrating our approach into a programming language, which would improve ease of use, readability, and program robustness. Additionally, there are many other data structures that could be made transactional with our system. Investigating both how best to adapt these datatypes for transactions, and what modifications to our system these may require are interesting questions. Further investigation of user-defined datatypes also seems warranted, for instance allowing higher-level operations like increments (that need not conflict with each other). Investigating how our system interacts with hardware transactional memory is another interesting problem. We hope that HTM can be implemented directly in our system, providing an immediate hybrid implementation.

8.2 Conclusion

This thesis has described a new way of supporting programmers who need to provide high-performance applications that can take advantage of the concurrency available on multi-core machines. Our approach allows application code to be organized as transactions; this is desirable because it simplifies reasoning about program correctness and removes many of the concurrency bugs that plague such code today.

We generalize STM by leveraging the fact that our transactions are implemented in software; this allows us to exploit abstraction. Application developers in our system mostly rely on efficient transaction-aware datatypes, such as trees and maps, that are available in a program library; they populate these data structures with their own data, typically packaged in a special transaction-aware type (`TBox`) that we provide. The result is that applications are both easy to build and perform well. Implementing the transaction-aware datatypes, however, is done by experts; this job is not easy, but we believe it is not much more difficult than what is required to implement highly-concurrent data structures that are not transaction-aware. Our results show we can outperform earlier systems by as much as a factor of 5 and provide better scalability.

Bibliography

[1] ACME Laboratories. thttpd - tiny/turbo/throttling HTTP server. URL http://acme.com/software/thttpd/.

[2] Bit-tech.net. IBM releases "world's most powerful" 5.5GHz processor. Retrieved from http://www.bit-tech.net/news/hardware/2012/08/29/ibm-zec12/1, 8 September 2012.

[3] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: High-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 6–15, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835703. URL http://doi.acm.org/10.1145/1835698.1835703.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM PPoPP Symposium*, Bangalore, India, 2010.

[5] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 187–200, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. doi: 10.1145/2628071.2628086. URL http://doi.acm.org/10.1145/2628071.2628086.

[6] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 56–67, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. doi: 10.1145/1229428.1229441. URL http://doi.acm.org/10.1145/1229428.1229441.

[7] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40, 2008.

[8] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. doi: 10.1145/1693453.1693464. URL http://doi.acm.org/10.1145/1693453.1693464.

[9] 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 Comput. Archit. News*, 39(1):39–52, Mar. 2011. ISSN 0163-5964. doi: 10.1145/1961295.1950373. URL http://doi.acm.org/10.1145/1961295.1950373.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. doi: 10.1145/1168857.1168900. URL http://doi.acm.org/10.1145/1168857.1168900.

[11] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, pages 185–192. IEEE, 2010.

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

[13] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proceedings of the 23rd International Conference on Distributed Computing*, DISC'09, pages 93–107, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 3-642-04354-2, 978-3-642-04354-3. URL http://dl.acm.org/citation.cfm?id=1813164.1813180.

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

[15] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic scalable atomicity via semantic locking. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 31–41, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. doi: 10.1145/2688500.2688511. URL http://doi.acm.org/10.1145/2688500.2688511.

[16] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345233. URL http://doi.acm.org/10.1145/1345206.1345233.

[17] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[18] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12), 2007.

[19] A. Hassan. *On Improving Transactional Memory: Optimistic Transactional Boosting, Remote Execution, and Hybrid Transactions*. PhD thesis, Virginia Polytechnic Institute and State University, 2014.

[20] A. Hassan, R. Palmieri, and B. Ravindran. Optimistic transactional boosting. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 387–388, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555283. URL http://doi.acm.org/10.1145/2555243.2555283.

[21] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345237. URL http://doi.acm.org/10.1145/1345206.1345237.

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

[23] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 131–131, New York, NY, USA, 2002. ACM. ISBN 1-58113-485-1. doi: 10.1145/571825.571847. URL http://doi.acm.org/10.1145/571825.571847.

[24] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM. ISBN 1-58113-708-7. doi: 10.1145/872035.872048. URL http://doi.acm.org/10.1145/872035.872048.

[25] Intel Corporation. Transactional Synchronization in Haswell. Retrieved from http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/, 8 September 2012.

[26] J. Jannotti and K. Pamnany. Safe at any speed: Fast, safe parallelism in servers.

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

[28] M. Lesani and J. Palsberg. Decomposing opacity. In *Proceedings of the 28th International Symposium, DISC 2014*, 2014.

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

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

[31] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. 4th Int'l Symp. on Workload Characterization (IISWC 2008)*, 2008.

[32] 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 *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 68–78, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. doi: 10.1145/1229428.1229442. URL http://doi.acm.org/10.1145/1229428.1229442.

[33] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, June 2007.

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

[35] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 76–86, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. doi: 10.1145/2688500.2688506. URL http://doi.acm.org/10.1145/2688500.2688506.

[36] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC '13*, Denver, Nov. 2013.

[37] M. Zhang, J. Huang, M. Cao, and M. D. Bond. Low-overhead software transactional memory with progress guarantees and strong semantics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 97–108, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. doi: 10.1145/2688500.2688510. URL http://doi.acm.org/10.1145/2688500.2688510.