CCSTM: A Library-Based STM for Scala

Nathan G. Bronson Hassan Chafi Kunle Olukotun

Computer Systems Laboratory
Stanford University
{nbronson, hchafi, kunle}@stanford.edu

Abstract

We introduce CCSTM, a library-based software transactional memory (STM) for Scala, and give an overview of its design and implementation. Our design philosophy is that CCSTM should be a useful tool for the parallel programmer, rather than a parallelization mechanism for arbitrary sequential code, or the sole synchronization primitive in a system.

CCSTM expresses transactional reads and writes as explicit method calls on instances of a reference type. Scala's flexible method names, implicit parameters, and closures keep the syntax concise, and the reference instances provide a natural way to express additional STM functionality. We use a novel hybrid of static and dynamic transaction scoping to retain composability while avoiding the barrier overheads that would otherwise result from an implementation as an unprivileged library. Experiments show that CCSTM's performance and scalability are on par with bytecode rewriting STMs.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel programming; D.4.1 [*Operating Systems*]: Process Management – Concurrency; Synchronization; Threads

General Terms Algorithms, Languages

Keywords Transactional memory, Scala

1. Introduction

The proliferation of multi-core processors means that more programmers are being thrust into the difficult world of shared memory multi-threading. Software transactional memory (STM) provides a compelling alternative to locks for managing access to shared mutable state; STM's declarative atomic blocks are free from deadlock, are composable, and do not require elaborate fine-grained decomposition to yield scalability.

In this paper we describe the design of CCSTM, a library-based STM for Scala. CCSTM deliberately sidesteps many of the semantic difficulties common in software implementations of transactional memory, by limiting its focus. We view CCSTM as a domain specific language (DSL) for use by parallel programmers that wish to build algorithms and data structures using optimistic concurrency control. CCSTM is not a drop-in replacement for locks, an all-encompassing concurrent programming model, or a mechanism for automatic parallelization of arbitrary code.

The most fundamental design choice for CCSTM was the decision to implement it entirely as a Scala library. Unlike STMs that transparently instrument all loads and stores of shared mutable state, accesses in CCSTM are explicit method calls on a Scala `trait` *`Ref`*. We refer to the resulting STM as 'reference-based', because memory locations managed by the STM are accessed only through an additional level of indirection.

While a reference-based STM adds one or two characters of program text to basic loads and stores, it leads to a safer and more full-featured interface. The encapsulation of transactionally-managed data allows CCSTM to provide strong atomicity and isolation with no performance impact on code that doesn't use the STM. In addition, *`Ref`* instances provide a first-class entity that names a memory location, which enables CCSTM to provide additional functionality to the user in a natural way.

CCSTM departs from the dynamic transaction scoping typical of STMs, using a hybrid approach. *`Ref`*'s methods locate the transaction via an implicit parameter of type *`Txn`*, which must be part of the lexical scope during compilation; this avoids the overhead of a dynamic lookup for most calls to the STM. Nesting of atomic regions, however, is resolved dynamically using a *`ThreadLocal`*; this avoids the need to add an implicit *`Txn`* parameter to every method called inside a transaction.

In this paper:

1. We describe CCSTM, a reference-based STM for Scala. CCSTM focuses on helping parallel programmers build optimistically concurrent algorithms and data structures, while restricting itself to implementation techniques that do not interfere with components of the system that do not use it (Section 3).
2. We show how a hybrid of static and dynamic transaction scoping can be used to reduce the performance penalty of a library-based STM while retaining ease of use (Section 3.3).
3. We introduce `unrecordedRead`, an STM primitive that relaxes read atomicity while allowing manual validation (Section 4.1).
4. We briefly present CCSTM's implementation, including a novel optimization for isolation write barriers (Section 6).
5. We demonstrate that although CCSTM is an unprivileged library, its performance is comparable to JVM STMs that use bytecode rewriting (Section 7).
6. We summarize some of the discussions that led from the original design goal to the current syntax. We point out the parts that work well and the parts that are cumbersome, and hypothesize about ways to address the latter (Section 8).

2. Motivation

An experimental feature such as software transactional memory should strive to impose only negligible costs on code that does not use it. Runtime performance costs are the most obvious, but extra complexity in the compiler, libraries, and language rules should also be minimized. A pay-as-you-go philosophy facilitates incremental adoption, it allows multiple implementations to coexist, and it reduces the penalty for failure.

One popular and reasonable interface design for transactional memory is to mimic lock-based critical regions. Users of such an

```
class Account(initialBalance: Money) {
  private var _balance = initialBalance

  def balance: Money = _balance

  def deposit(m: Money) {
    assert(m >= 0)
    _balance += m
  }

  def withdraw(m: Money) {
    assert(m >= 0)
    if (_balance < m)
      throw new OverdraftException
    _balance -= m
  }
}

object Account {
  def transfer(src: Account, dst: Account, m: Money) {
    src.withdraw(m)
    dst.deposit(m)
  }
}
```

Figure 1. Code that performs an account transfer without any locking or other concurrency control.

```
class Account(initialBalance: Money) {
  private val _balance = Ref(initialBalance)

  def balance: Source[Money] = _balance

  def deposit(m: Money)(implicit t: Txn) {
    assert(m >= 0)
    _balance := _balance() + m
  }

  def withdraw(m: Money)(implicit t: Txn) {
    assert(m >= 0)
    if (_balance() < m)
      throw new OverdraftException
    _balance := _balance() - m
  }
}

object Account {
  def transfer(src: Account, dst: Account, m: Money) {
    STM.atomic { implicit t =>
      src.withdraw(m)
      dst.deposit(m)
    }
  }
}
```

Figure 2. One way to implement the atomic balance transfer function using CCSTM. This code uses `apply()` and `:=` operators for performing transactional reads and writes, and expresses the atomic block as an anonymous *`Txn => Unit`*.

STM declare the beginning and the end of an atomic block, and all memory accesses that occur within the dynamic scope of the block are transparently redirected to the STM. For a VM language like Scala this redirection can be introduced by the VM's JIT, by bytecode rewriting at class load time, or during the initial compilation of the high-level language. The dynamic scoping of such an approach, however, means that it is generally not possible to limit instrumentation to only classes that are used in an atomic block. An STM that is deeply integrated into the VM's JIT can minimize the performance and code bloat impacts of the instrumentation by performing it citation lazily, but the engineering effort to add this support to a production VM is prohibitively large. Instrumentation of the bytecode at compilation or class loading has the lowest engineering cost, but results in two copies of each method. This is the strategy adopted by the Multiverse [18] and Deuce STM [12] STMs for the JVM. While this cost may eventually be considered acceptable, it places a high hurdle to integration into Scala's standard library. An additional drawback of bytecode rewriting is that it is not composable. If two modules use different STMs then they cannot be used in the same program.

The alternative approach adopted by CCSTM is to require the programmer to perform explicit calls to the STM. While less convenient for simple uses, this limits performance side-effects on code that does not use atomic blocks, and it allows the STM to be constructed entirely as an unprivileged library. When coupled with an STM design that does not assume it is managing all threads, the result is a pay-as-you-go transactional memory suitable for experimentation and incremental adoption.

Scala's flexible syntax makes a library-based STM tractable. Operator overloading makes transactional loads and stores concise, and implicit parameters allow the current transaction context to be statically threaded through the code without explicitly including it in each call. The resulting STM can be considered an embedded DSL for optimistic concurrency.

3. The Basic Interface

As a recurring example, consider a class that encapsulates the balance of a checking account[1]. Absent any concurrency control, we might write the code in Figure 1. Adding pessimistic concurrency control to this code by locking accesses to *`Account`* instances is not straightforward, because both the source and destination account must be locked during a `transfer`. Unless a global lock order is followed this can easily lead to deadlock. CCSTM allows the atomic balance transfer to be expressed easily, guaranteeing that both balance adjustments are performed atomically and without deadlock. Figure 2 shows one way to express this using CCSTM.

3.1 References – *Ref*[A] and *Ref.Bound*[A]

The most fundamental data type in CCSTM is *`Ref`*`[A]`, which mediates access to an STM-managed mutable value. Read-only methods are separated into a covariant *`Source`* trait and write-only methods are separated into a contravariant *`Sink`* trait. The current transactional context is passed during each method call via an implicit parameter. Reads and writes on a reference may be performed with the `get` and `set` methods, respectively, or with the more concise `apply()` and `:=` operators. Section 8.1 discusses the choice of method names in more detail.

Non-transactional access to the contents of a reference are provided by a view returned by `nonTxn`. This view implements methods that parallel those of the reference, but that don't require a *`Txn`*. We say that the view is *bound* to the non-transactional context, so the view trait is named *`Ref.Bound`*. Views may also be bound to a transactional context via *`Ref`*`.bind`. These bound references do not require a *`Txn`* parameter, but may only be used until the end of the transaction. Figure 3 shows the subclassing relationship between the traits that implement unbound and bound references, and some of their methods. The separation between *`Ref`*, *`Source`*, and *`Sink`*, and the operator syntax for accesses are modeled after Spiewak's Scala STM [17]. The *`Ref`* $\leftrightarrow$ *`Ref.Bound`* duality is unique to CCSTM, as far as is known by the authors.

Bound views for non-transactional access create a syntactic difference between transactional and non-transactional reads and writes. This allows the expert programmer to selectively relax isolation by performing a non-transactional access inside an atomic block, without requiring an escape action. The non-isolated access

[1] This example is adapted from Deuce STM's bank benchmark [12].

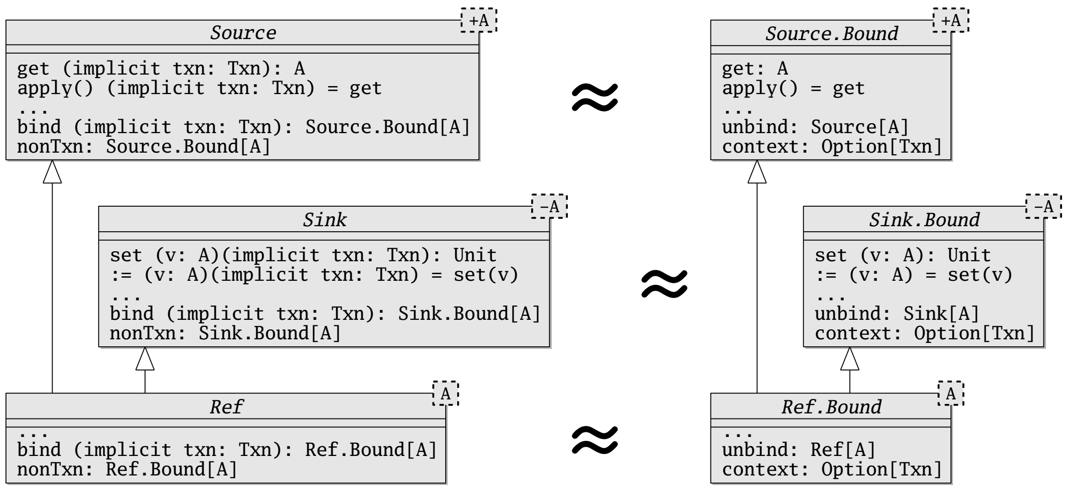

Figure 3. Traits that provide access to an STM-managed memory location. Transactional access can occur through either *Ref* or a *Ref.Bound* returned from *Ref*.bind, non-transactional access occurs through a *Ref.Bound* returned from *Ref*.nonTxn. *Source*[+A] and *Sink*[-A] decompose the covariant and contravariant operations of *Ref*[A].

is visually differentiated by including the token nonTxn. In Section 4.1 we will introduce unrecordedRead, a way of relaxing isolation for reads while retaining the ability to validate them.

3.2 Declaring and executing an atomic block

CCSTM's atomic blocks are functions with type (*Txn* => Z). Transactional execution is provided by passing a block to the atomic method of the STM object. This method creates or joins a transaction, passes the transaction context to the block, attempts to commit the transaction, and retries the block if the transaction could not be committed.

We have found two idiomatic ways to make the block's *Txn* argument available as an implicit value. For inline transactions, the most concise code adds the implicit modifier to a closure's parameter[2], as shown in Figure 2 on Lines 45 to 48. This syntax can be shortened even further if STM.atomic is imported. If the transaction's body consists only of a call to a method that threads the implicit *Txn* parameter, the method can be passed to STM.atomic by partially applying it. In this style the *Account*.transfer method might be decomposed into:

```
def transfer(src: Account, dst: Account, m: Money) {
  STM.atomic(transferInTxn(src, dst, m)(_))
}
def transferInTxn(src: Account, dst: Account,
      m: Money)(implicit txn: Txn) {
  src.withdraw(amount)
  dst.deposit(amount)
}
```

The decision to statically bind CCSTM's transactions to *Ref* invocations was made for performance reasons. To dynamically scope the transactions, the current transaction must be identified by each read and write barrier. This is an extremely frequent operation. Bytecode rewriting STMs have two options for efficiently performing this lookup: they can add a field to the system-wide *Thread* class, or they can weave a *Txn* parameter into the transactional version of every method. A library-based STM running on the JVM must restrict itself to *ThreadLocal*, which navigates from the *Thread* to a thread-local hash table, and from there to the dynamically scoped value. We do not include a detailed experimental comparison, but we have observed that the cost of a single *ThreadLocal* lookup can increase the per-read cost by up to 50% over performing statically-bound reads in a large transaction.

[2] This syntax is new in Scala 2.8.

3.3 Dynamic scoping for nested transactions

Statically scoped transactions using an implicit *Txn* are invisible when making calls to *Ref*, but they add clutter when decomposing the work of an atomic block into methods. Each method that expects to be run in a transaction must declare the implicit parameter. This mechanism also makes it difficult to make methods that can be called from both inside and outside a transaction. For example, the expected behavior for a call to deposit from a non-transactional context is clear, but this won't be allowed. The worst pitfall of the purely-static approach comes if a transaction is active but is not available in the lexical scope, leading the user to create a new transaction that is not nested in the active one!

The solution is for *Ref*'s method to bind the *Txn* statically, while STM.atomic searches the thread's dynamic scope for an active *Txn*. This means that a method that should be callable from either context may omit the implicit *Txn* parameter and create a new atomic block. When called from a transactional context the new atomic block will be nested or subsumed in the outer one. With this strategy deposit might be coded:

```
def deposit(m: Money) {
  assert(m >= 0)
  STM.atomic { implicit t =>
    _balance := _balance() + m
  }
}
```

The current CCSTM implementation flattens nested transactions, so when called from the atomic block in transfer this version of deposit will perform its work in the outer context. If called when no transaction is active on the thread (regardless of the static scope) it will create a new transaction.

4. Advanced Functionality

A CCSTM *Ref* provides both a value that names a specific memory location, and a namespace for operations beyond loads and stores.

4.1 Relaxed isolation

Some algorithms can benefit from transactional reads that are not guaranteed to be consistent, but that still observe speculative stores made by the current transaction. The inconsistent value may be used to make a heuristic decision, such as a hash table resize, algorithm-specific knowledge may be used to guarantee atomic behavior of the transaction despite a subsequent invalidation, as in early release when searching a binary tree, or life cycle callbacks may validate using specific knowledge.

Previous TM systems have provided several mechanisms for relaxing atomicity and isolation. Early release allows reads to be removed from the read set prior to commit [10]. Escape actions suspend the current transaction temporarily [6]. Open nested transactions allow the actions of a nested transaction to be committed in a non-nested fashion. CCSTM supports early release and escape actions for individual accesses. Escape action are implemented by simply using a `nonTxn` bound view from inside a transaction. Early release is supported in a principled manner by *`Source.Bound`*`.releasableRead`. This method returns a *`ReleasableRead`* instance that bundles the requested value with a method that removes the record of the access from the transaction's read set. This interface eliminates the danger that an algorithm will remove a read that it did not perform, but it still requires careful reasoning to guarantee correctness after the read has been released.

As an alternative to a releasable read, CCSTM includes a new abstraction, `unrecordedRead`. This method performs a transactional read, but instead of adding an entry to the read set it bundles the read's meta-data into an *`UnrecordedRead`* instance. The caller may then use this instance to manually validate that the returned value is still valid.

Like many STMs, CCSTM performs transactional reads by associating a version number with each managed memory location, recording the version prior to a transactional read, and checking during validation that the version number remains unchanged. An *`UnrecordedRead`* contains the read value and the prior version, but rather than automatically validating the read during commit, validation is exposed to the programmer via the method `stillValid`. An unrecorded read is considered to still be valid if the only changes that have been made to the referenced memory location were performed by the read's transaction. This definition also produces a meaning for unrecorded reads of the `nonTxn` bound view: `stillValid` will return true only if no change has been made to the managed value. This leverages the STM's meta-data to solve the ABA problem[3].

4.2 Semantic conflict detection for reads

Unrecorded reads can be paired with life cycle callbacks to implement Abstract Nested Transactions [9]. For the simple case where a single transactional read is modified by an idempotent function, *`Ref`* provides `map[Z](f: T => Z): Z`. A transactional call to `x.map(f)` returns the same value as `f(x.get)`, but no rollback is triggered by a conflicting write to `x` if the result of the mapping does not change. Without this semantic conflict detection, the STM must initiate rollback any time `x` is changed concurrently, even if that change is masked by the application of `f`.

Consider the branch taken by Figure 2's Line 37 during an attempt to withdraw 1,000 *`Money`* from an account with a balance of 500. At this point `_balance` will be included in the transaction's read set, so a concurrent deposit of 100 will cause the withdrawal transaction to be rolled back despite not changing the withdrawal's outcome. Although only a single bit of information about the balance was retained, the STM must conservatively assume that any

[3] The ABA problem is when an observer falsely concludes that a value has not changed, because the watched value went from A to B, then back to A.

```
class Ref[T] {
  ...
  def map[Z](f: T => Z)(implicit t: Txn): Z = {
    val u0 = unrecordedRead
    val result = f(u0.value)
    t.addReadResource(new Txn.ReadResource {
      var u = u0 // latest unrecorded read

      def valid(t2: Txn) = {
        if (u.stillValid) {
          true
        } else {
          // reread and compare to original
          u = unrecordedRead
          (result == f(u.value))
        }
      }
    }, 0, false)
    result
  }
}
```

Figure 4. *`Ref`*`.map` implemented by `unrecordedRead` and a *`ReadResource`* callback. The callback is invoked during read set validation. Conflicting changes to the reference do not require the transaction to be rolled back if `f(get)` does not change.

change invalidates the speculative execution. If we move the inequality application into a predicate applied by `map`, however, the STM can recompute that bit and determine that the withdrawal transaction is still valid:

```
// if (_balance() < m) ...
if (_balance.map(_ < m)) ...
```

By allowing the programmer to express more of her intention to CCSTM, `map` can avoid rollbacks and lead to better scalability, especially for transactions that have already performed a substantial quantity of work. Figure 4 shows how `map` may be implemented using `unrecordedRead` and a validation handler.

4.3 All of the ways to read and/or write

What follows is the complete list of the access operations provided by *`Ref.Bound`*. Many of these methods have equivalents in *`Ref`* that take an implicit *`Txn`*, although to reduce the API's surface area some methods are not mirrored. Every access operation in *`Ref`* is present in *`Ref.Bound`*. All of the methods are defined for both transactional and non-transactional contexts, even if they are mainly useful only for one of those. *`Source.Bound`* declares methods that read, *`Sink.Bound`* declares methods that write, and *`Ref.Bound`* declares methods that simultaneously read and write. Several of the methods that read and write are both more concise and more efficient than a simple transaction.

`Source.Bound:`

`apply(): T`
: Equivalent to `get`.

`get: T`
: Reads the value managed by the bound *`Ref`*. If this view is bound to a non-transactional context, the read will be strongly atomic and isolated with respect to all transactions, and will linearize before returning.

`map[Z](f: T => Z): Z`
: Returns `f(get)`, possibly reevaluating `f` to avoid rollbacks (`f` must be idempotent).

`await(p: T => Boolean)`
: Blocks until `p(get)` is true. Transactional contexts block by rolling the transaction back using `retry`, the modular blocking primitive. Non-transactional contexts just block.

`unrecordedRead: UnrecordedRead[T]`
: Returns an instance that wraps the value that would be returned by `get`, but does not add anything to the transaction's read set (if any).

`releasableRead: ReleasableRead[T]`
: Reads the value managed by the bound *Ref*, and returns that value in an instance that allows the corresponding read set entry (if any) to be removed.

Sink.Bound:

`:=(v: T)`
: Equivalent to `set(v)`.

`set(v: T)`
: Updates the value managed by the bound *Ref*. If this view is bound to a non-transactional context, this method will linearize the store before returning.

`tryWrite(v: T): Boolean`
: Immediately performs an update and returns true, or does nothing and returns false.

Ref.Bound extends Source.Bound with Sink.Bound:

`readForWrite: T`
: Returns the same value as that returned by `get`, but adds the bound *Ref* to the write set of the transaction context, if any.

`getAndSet(v: T): T`
: Atomically invokes `set(v)` and returns the old value.

`compareAndSet(b: T, v: T): Boolean`
: Atomically performs `(b == get) && { set(v); true }`

`compareAndSetIdentity(b: T, v: T): Boolean`
: Atomically performs `(b eq get) && { set(v); true }`

`weakCompareAndSet(b: T, v: T): Boolean`
: Either performs `compareAndSet` or returns false.

`weakCompareAndSetIdentity(b: T, v: T): Boolean`
: Either performs `compareAndSetIdentity` or returns false.

`transform(f: T => T)`
: Atomically replaces the stored value v with `f(v)`.

`getAndTransform(f: T => T): T`
: Atomically replaces the value `v` stored in the *Ref* with `f(v)`, returning the old value.

`tryTransform(f: T => T): Boolean`
: Immediately atomically transforms this reference and returns true, or returns false.

`transformIfDefined(pf: PartialFunction[T,T]): Boolean`
: Atomically replaces the value `v` stored in the bound *Ref* with `f(v)` if `pf.isDefinedAt(v)`, returning true, otherwise leaves the value unchanged and returns false.

4.4 Conditional retry

CCSTM supports the `retry` and `orElse` primitives introduced by Harris et al. in Haskell's STM [7], although the current lack of partial rollback when nesting makes them less expressive than the original. The `retry` primitive causes the surrounding transaction to be rolled back, but retry is postponed until at least one of the values read by the transaction has changed. `orElse` combines two transactions, attempting the second if the first calls `retry`, then blocking both transactions if the second calls `retry`. Intuitively, a call to `retry` is a dead end; the STM will restart the transaction only after it might take a different path. Similarly, `orElse` composes two alternatives that are each satisfactory, and requests that whichever one can avoid the dead end should be executed.

Currently, CCSTM encodes `retry` as a method of the `STM` object, and combines composition and execution of atomic blocks into `STM.atomicOrElse[Z](blocks: (Txn => Z)*): Z`. While we have experimented with an implicit conversion from `(Txn => Z)` to an *AtomicBlock* that provides a rich interface, we have not yet found a syntax that works well. If `retry` is used without `orElse`, then the normal `STM.atomic` method may be used.

As a (hopefully) contrived example, the bank could use modular blocking to withdraw money from exactly one of a number of accounts, blocking until success:

```
class Account {
  ...
  def withdrawOrRetry(m: Money
        )(implicit t: Txn) {
    if (_balance() < m) STM.retry
    _balance := _balance() - m
  }
}
object Account {
  def withdrawFromSomeone(m: Money, srcs: Account*) {
    val blocks = srcs map { s =>
        { (t: Txn) => s.withdrawOrRetry(m)(t) } }
    STM.atomicOrElse(blocks: _*)
  }
}
```

5. Semantics

CCSTM provides strong semantic guarantees for the memory locations that it manages, but does not attempt to hide the fact that transactions may be executed more than once. All accesses to *Ref* instances are strongly isolated and atomic, and transactions guarantee opacity. CCSTM does not handle or prevent irrevocable actions inside transactions. Instead, it provides a rich set of life cycle callbacks that allow a variety of strategies to be implemented.

5.1 Strong isolation

One of the benefits of the reference-based approach is that it avoids isolation problems between transactional and non-transactional accesses to the same memory location, without requiring any changes to the underlying type system.

At its most basic, a software transactional memory is a way of isolating a group of memory accesses and verifying that those accesses are equivalent to some serial execution. The STM barriers that perform the transactional reads and writes include code that blocks or rolls back any accesses that violate atomicity or isolation. If non-transactional code bypasses the barriers and accesses an STM-managed memory location directly, however, the barriers can no longer detect all violations.

There are three potential responses to the weak isolation between direct memory accesses and concurrent transactions:

- The runtime can provide strong isolation and atomicity by redirecting all memory accesses to barriers, even non-transactional accesses. While there has been some research in using dynamic recompilation to reduce the performance penalty of strong isolation, these require either deep integration with the VM's JIT [16] or a substantial warmup period [1].
- The language can declare that a conflicting concurrent access from both inside and outside a transaction is an error. This doesn't sound too onerous, but the optimistic nature of transactions means that failed speculations must also be considered: inconsistent transactions may execute conflicting accesses from an impossible branch, or they may execute conflicting accesses after they have become doomed. Restrictions on commit order can prevent some of the most surprising behaviors [13], but the resulting systems still require whole-program reasoning to guarantee correctness. The privatization and publication problems refer to isolation failure for specific idioms.

- The type system can prevent direct access to any memory location that might be touched transactionally [14]. This can take the form of extending the type and access rules on normal mutable memory locations, or of encapsulating transactionally-managed data as private variables of some sort of cell, as in Haskell [8] and Clojure [11]. We refer to the latter approach as a reference-based STM.

Scala favors safety and compile-time checking of program correctness, so the authors are of the opinion that it is only natural to employ types to avoid the problems of weak isolation. In the long term, an extension to Scala's types seems possible, but in the short term a reference-based approach seems the most practical. CCSTM provides strong isolation by encapsulating all transactionally-managed memory locations inside references.

5.2 Opacity

A subtle issue with STM is that, unless special care is taken, only committed transactions are guaranteed to be consistent. Speculative transactions may observe an inconsistent state and only subsequently detect that they should roll back. These 'zombies' can produce surprising behavior by taking impossible branches or performing transactional accesses to the wrong object. This problem is greatly magnified in a reference based STM, because the STM cannot provide a sandbox that isolates all actions taken by the zombie. The read of a single impossible value may produce an infinite loop, so a transparent STM must either prevent inconsistent reads or instrument back edges to periodically revalidate the transaction. Only the first option is available to an STM implemented as a library.

The TL2 [2] and LSA [15] algorithms use a global time-stamp to efficiently validate a transaction after each read, guaranteeing consistency for all intermediate states. This correctness property is formalized as *opacity* [5]. CCSTM is based on SwissTM [3], which adds eager detection of write-write conflicts to TL2's validation algorithm.

5.3 Irrevocable actions and structural conflicts

One of the side effects of CCSTM's alternate syntax for transactional barriers is that it avoids creating the impression that the STM can magically parallelize all existing sequential code, or that atomic blocks are always a better replacement for locks. There are both semantic and practical reasons why this is not the case, even for STMs with deep integration into the VM.

The semantic problems with hiding rollback and retry come from actions that the STM cannot isolate or undo, such as I/O or calls to external libraries. CCSTM does not try to automatically handle irrevocable actions. Instead, it provides handlers that allow user code to implement a variety of strategies. Five types of callbacks may be registered with a transaction:

- *before-completion* – invoked before the transaction attempts to commit, regardless of whether it is already doomed;
- *read-resource* – invoked each time the transaction's read set is validated (CCSTM can avoid validation for most read-only transactions);
- *write-resource* – participates in a two-phase commit, voting on the outcome and then receiving the consensus decision;
- *after-commit* – invoked after the transaction has committed, but before the application has been informed of the success; and
- *after-rollback* – invoked after the transaction has rolled back, but before it is retried or failure is reported to the application.

The practical problem with executing code that was not designed to be executed inside an atomic block is that such code often contains incidental shared accesses that the STM must treat as conflicts. An example of this is the size field of a collection, which is often accessed by every mutating operation. Unless care is taken to distribute this variable over multiple memory locations, no concurrency will actually be available.

CCSTM provides several mechanisms for reducing transaction conflicts with semantic conflict detection. A sophisticated user can combine the `releasableRead` or `unrecordedRead` primitive (Section 4.1) with life cycle callbacks to manually implement their own conflict detection. For simple cases, *`Ref`*`.map` (Section 4.2) makes it trivial to use Abstract Nested Transactions (ANTs) to avoid rollback [9]. For the special case of contention on integer values, CCSTM includes *`LazyConflictIntRef`*, which uses ANTs for all inequality comparisons, increments, and decrements, and *`StripedIntRef`*, which is optimized for low-contention increment and decrement with occasional reads. Both of these classes implement *`Ref`*`[`*`Int`*`]`.

6. Implementation

CCSTM's version management and conflict detection use the SwissTM algorithm [3]. Version management is lazy, but write permission is acquired eagerly. Time-stamps are allocated 51 bits, making CCSTM effectively immune from counter overflow.

6.1 Meta-data indirection

Meta-data for a managed memory location consists of a single `long`. It is assumed that each memory location maps to a unique meta-data value, but not vice versa. This allows objects with multiple fields to use a single piece of meta-data, and it allows arrays to choose a variety of granularities of conflict detection. While some optimizations are possible for situations where the data-to-meta-data mapping is one-to-one, in informal experiments we found that the benefits were smaller than the additional indirection costs.

*`Ref`*s perform their accesses to both data and meta-data through methods of an internal trait called a *`Holder`*. This indirection allows multiple storage strategies to be easily provided, which can yield an important reduction in the number of live objects in the VM. For example, if the static or manifest type of the initial value is known to be an `Int`, then the *`Ref`* factory method will return a reference whose holder stores the value in an unboxed form. As a more extreme example, CCSTM provides a transactional array-like class that internally uses one array for values and one array for meta-data, eliminating the n intermediate objects that would be required by an *`Array`*`[`*`Ref`*`[A]]`.

6.2 Global time-stamp optimizations

To reduce contention on the shared time-stamp, CCSTM uses TL2's GV6 scheme [2]. This mechanism is based on the observation that, while committed values must be given a time-stamp later than the version clock that was present at the beginning of the commit, it is not required that the global clock is actually advanced. Advancing the global clock reduces the need for validation in later transactions, but when many threads are using the STM, this goal is satisfied even if only a fraction of transactions attempt to advance the current time.

CCSTM performs a novel additional optimization to reduce the overhead of non-transactional accesses. Unlike a transaction, a solitary strongly-isolated read or write in a TL2-style STM does not need to sample the global clock to provide opacity. This means that we can allow a sequence of non-transactional writes to advance a reference's time-stamp to an arbitrary point in the future, without advancing the global time-stamp. If a transaction attempts to read such a far-future value it handles it via the normal GV6 mechanism, by advancing the global time-stamp and then revalidating. To limit the potential impact of these booby-trapped references, we only allow non-transactional writes to advance time-stamps a limited distance into the future. Even a small window (CCSTM defaults to 8) dramatically reduces contention on the global time-stamp.

6.3 Avoiding starvation

Optimistic concurrency control is vulnerable to the *starving elder* problem, in which a large transaction can never be committed because it is continually violated by small transactions. CCSTM uses a simple contention management scheme to prevent this. Each execution attempt is assigned a random priority that is used to resolve write-write conflicts. If a transaction has not yet begun to commit, then a higher priority transaction may doom it and steal its locks. In addition, transactions that have already failed several times enter a 'barging' mode in which they acquire write permission during reads. The result is that even large transactions will eventually succeed, because they will eventually receive the highest priority in the system.

6.4 Polite blocking

An important design goal for CCSTM is support for incremental use inside a larger application. This means that busy waiting or exponential back-off are not suitable mechanisms for blocking. Many STMs target parallel speedups for only CPU-bound applications, and so assume that they own all threads and perform all synchronization. CCSTM makes neither of these assumptions, taking care to block using the normal synchronization primitives of the underlying VM.

Blocking may be required to obtain write permission, or because of an explicit use of the `retry` primitive. Writers and waiters must agree on a condition variable that will be used to signal that the waiter should re-attempt whatever action led to their choice to block. If the set of condition variables is too small, there will be many spurious wakeups. If the set is too large, then transaction commits may need to perform a large amount of extra work.

Accesses that are blocked by another transaction await notification on the *`Txn`* instance itself. No such instance is available for threads blocked by a non-transactional write, or that are performing a conditional retry, so the system also maintains 64 lists we refer to as 'wakeup channels'. These channels contain a list of pending wakeups, which are single-shot gates (similar to a Java *`CountDownLatch`* with a count of 1). Each memory location is associated with a wakeup channel by hashing its identity. To await the modification of a memory location, a thread enqueues a new pending wakeup instance, sets a 'wakeup pending' bit in the location's meta-data, rechecks the blocking condition, and then puts itself to sleep on the gate. If an update notices the wakeup pending bit, it triggers and removes all of the pending wakeups for the corresponding channel. A thread may wait on multiple memory locations simultaneously by enqueuing its pending wakeup instance to multiple channels. The choice of 64 wakeup channels makes it easy to accumulate the effects of a transaction in a `long`. If a system makes extremely heavy use of the `retry` mechanism by having many blocked threads, a larger number of channels might be appropriate.

6.5 JVM versus CLR

Scala is designed to target both the JVM and the CLR virtual machines. In its current implementation, CCSTM uses the *`Atomic*`* classes from *`java.util.concurrent`* to perform atomic compare-and-swaps and volatile array accesses. The authors are not experts on the CLR memory model, but we believe that it would be straightforward to retarget CCSTM to the CLR by using methods in *`System::Threading::Interlocked`*.

7. Performance

CCSTM's implementation as an unprivileged library introduces several overheads when compared to a bytecode rewriting STM or modified VM: *`Ref`* adds a level of indirection; JVM erasure adds boxing overheads for *`Ref`*`[T]` when `T` is a primitive type; avoiding boxing for long-term storage requires that the underlying memory locations be accessed through virtual methods; dynamic scoping involves a hash table lookup, either implicitly inside *`ThreadLocal`* or explicitly with a *`Thread`* key; and low-level atomic operations performed by the STM cannot use the unchecked primitives in `sun.misc.Unsafe`. The actual impact of these overheads, however, is reduced or eliminated by the compiler optimizations of a modern JVM and the out-of-order superscalar pipeline of a modern processor.

To verify that CCSTM's library-based design does not impose a prohibitive performance penalty, we compared it to Deuce STM and Multiverse, STMs for the JVM that perform bytecode rewriting during class loading [12, 18]. We performed a direct encoding of Deuce STM's bank benchmark into Scala+CCSTM, and compared this version to the Java original running under the bytecode rewriting STMs. (While the example code in this paper uses an immutable *`Money`* numeric type, the evaluated benchmark uses 32-bit floating point values like the original.) Deuce STM provides two algorithms, TL2 and LSA, each of which has optional contention management (CM). For each configuration and thread count, we report the throughput of a Deuce STM algorithm as the maximum of the throughput with no CM or with CM (Polka for TL2, Timestamp for LSA). For almost all configurations we tested, CM reduced throughput. CCSTM and Multiverse were tested using their default configuration. The bank benchmark includes its own harness, which we configured so that no overdrafts were triggered. We used a 20 second warmup, and then measured the number of transactions committed during 10 seconds, averaging across three invocations of the JVM.

Experiments were run on a Dell Precision T7500n with two quad-core 2.66Ghz Intel Xeon X5550 processors, and 24GB of RAM. Hyper-Threading was enabled, yielding a total of 16 hardware thread contexts. We used Scala version 2.8.0.Beta1. We ran our experiments in Sun's Java SE Runtime Environment, build 1.6.0_16-b01, using the HotSpot 64-Bit Server VM with dynamic escape analysis and compressed object pointers enabled. Deuce STM was version 1.3.0. Multiverse was version 0.4.

For the low-contention experiment (Figure 5) we set the number of accounts to 64 times the number of threads. For the high-contention experiment (Figure 6) we set the number of accounts to the number of threads. Single-threaded execution is not included in the high-contention setup, as it has no contention. Because at most 16 threads are executing at any time, high-contention runs have fewer conflicts at 32 and 64 threads than for lower thread counts, and so can continue to scale so long as blocked threads do not consume too many resources. This effect is present to a lesser degree for the low-contention runs, where CCSTM's throughput continues to rise gradually as its rollback rate drops from 1.0% at 16 threads to 0.5% at 32 and 0.3% at 64.

For low contention configurations, Deuce STM and CCSTM have similar performance and scalability, so long as the multi-threading level is less than or equal to one. Under high contention Deuce STM substantially outperforms CCSTM, so long as each thread gets its own hardware context. This is because Deuce STM never yields, sleeps, or blocks, which avoids the cost of context switches. At thread counts 32 and 64, however, threads must share processing resources and this strategy results in a catastrophic performance drop off. Multiverse uses an exponential back off algorithm using `Thread.sleep`. This approach handles high multi-threading levels better than busy-waiting, but results in extra con-

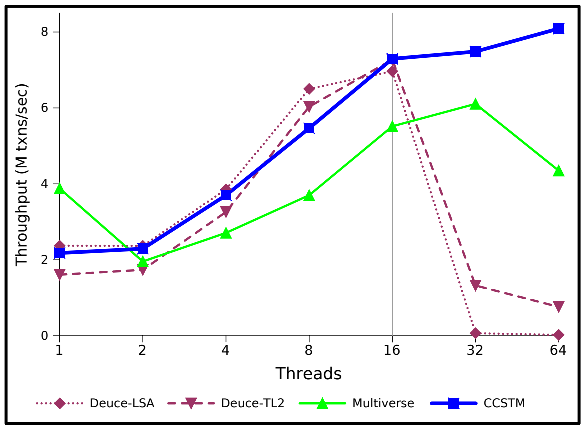

Figure 5. Throughput for the bank benchmark in a low contention scenario, on a machine with 16 hardware thread contexts. The number of accounts is 64 times the number of threads.

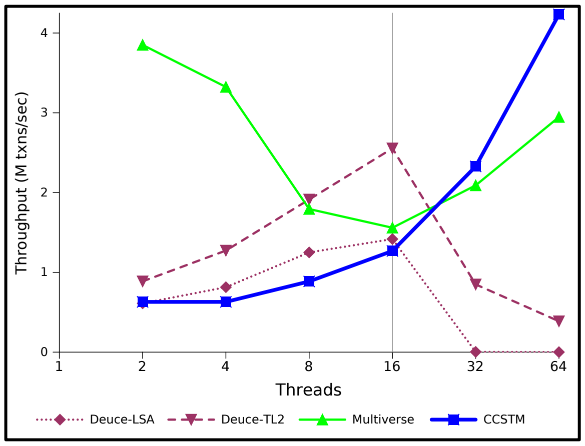

Figure 6. Throughput for a high contention scenario. The number of accounts is equal to the number of threads. Each transaction touches two accounts.

text switches as threads awake to recheck their blocking condition. Sleeping works very well for low thread counts of the high-contention configuration, because it leads to intervals of uncontended single-threaded execution. CCSTM's blocking synchronization implementation is the most expensive at low thread counts, but it yields a robust scaling curve.

The three STMs have different algorithms, feature sets and engineering tradeoffs, so these experiments do not allow us to exactly measure the overhead imposed by the library-based design. They do demonstrate, however, that the practical overheads that exist are small enough to be tolerable. We leave as future work a breakdown of the costs inside CCSTM.

8. Discussion and Future Work

STM research is mature enough that the most difficult design decisions for CCSTM were all found in the interface.

8.1 Read barrier syntax

The most difficult syntactic choice was the method name for a transactional read. Four alternatives were considered:

1. `elem` – A pair of methods `elem` and `elem_=` could be used to provide the illusion that transactional data was actually stored in a variable of *Ref*. This approach is self-consistent, but is much too verbose.
2. An implicit conversion from *Ref*`[A]` to `A` – This is the most concise, but it interferes with further implicit conversions, and with type inference for generic methods or data structures. If `x` is a *Ref*[*String*] holding `"foo"`, for example, `("foo" == x)` would return false.
3. `unary_!` – A unary operator is the most concise explicit way of denoting a read, and prefix forms of these are the only ones that don't trigger Scala's line merging heuristic. Initially we settled on a `!` prefix for reads. This works well for arithmetic expressions, but it can be confusing when used in a conditional test, as in `(!x == "foo")`. It also does not chain well, requiring extra parentheses: `(!x).length`. Because of these problems, we found ourself often reverting to the more verbose forms `(x.get == "foo")` and `x.get.length`.
4. `apply()` – This is the only operator-like method that can safely occur at the end of the line, which allows it to be placed after the expression that produces the *Ref* to read. This means that it chains properly in complex operations. We initially avoided using `apply()` for read barriers, because inclusion of optional parentheses on a method call is often used in Scala to draw attention to side effects. We decided that using `()` in read barriers was warranted, however; although read barriers don't perform any visible mutation, their access to shared mutable state does require extra care. The current CCSTM code base uses `apply()` for concise transactional reads.

For fields that are almost always accessed inside an atomic block, we sometimes found it convenient to create transactional accessor methods. The full *Ref* is published via a longer name for non-transactional or advance operations, while the basic property name is available within a transaction using Scala's basic field syntax:

```
class Node {
  val nextRef: Ref[Node] = ..
  def next(implicit txn: Txn): Node = nextRef()
  def next_=(v: Node)(implicit txn: Txn) {
    nextRef := v
  }
}
```

8.2 Dynamically scoped non-transactional reads

The bound view returned by *Ref*`.nonTxn` makes non-transactional reads and writes explicit. This makes those accesses visible in the code, and allows the compiler to statically check that the remaining transactional accesses only occur within the static scope of a *Txn*. This also allows the implementation of escape actions. An expert user may use the nonTxn view inside a transaction to perform reads and writes that bypass an active transaction context. The verbosity of explicit calls to nonTxn makes non-transactional accesses inconvenient, but this may actually be beneficial if it discourages premature optimization using relaxed isolation.

The main disadvantage that we have observed is that code composability is hindered by the escape action semantics of nonTxn. The advanced *Ref.Bound* operations such as `transform` and `getAndSet` (Section 4) allow some small transactions to be rewritten as a single operation, but this changes the semantics if there is an active transaction. This is similar to the problem that led us to use dynamic scoping for nesting (Section 3.3), but with less possibility to amortize the costs of the required *ThreadLocal* lookup. We may add a third binding mechanism that requests a transactional binding (like *Ref*`.bind`) if a dynamic lookup finds an active transaction or a non-transactional binding (like *Ref*.nonTxn) if none is found.

8.3 Partial rollback of nested transactions

We plan to extend CCSTM's implementation to support partial rollback. This is important to properly support composition with orElse, and it is also required to provide failure atomicity (rollback without retry after a user exception). The only changes to the CCSTM API required for partial rollback are for the life cycle callbacks, which must be extended to allow notification of the commit or rollback of a nested context prior to the completion of the top level transaction.

8.4 @specialized

The @specialized annotation has the potential to reduce boxing overheads for read and write barriers that access a primitive type [4]. CCSTM already avoids long-term boxing by using separate implementations of *Ref* for each primitive (selecting using a *ClassManifest* during creation if the type is not statically known), but boxing occurs during the reused portions of the barrier code. The current implementation of @specialized interacts non-trivially with the manual specialization of *Ref* that is already performed, but restructuring of CCSTM's concrete classes may enable performance improvements.

9. Conclusion

STM's high level programming model addresses many of the challenges of shared memory multithreading, but to be most useful its benefits should be provided in a pay-as-you-go fashion. CCSTM accomplishes this by providing transactional memory as a normal Scala library.

CCSTM uses Scala's features to embed STM as a DSL, rather than using bytecode rewriting or VM modifications to transparently redirect loads and stores. Its syntax is concise, and its performance is on par with bytecode rewriting STMs. The references that encapsulate transactionally-managed memory locations add some clutter to the user's code, but they also provide a natural way for the programmer to take advantage of more sophisticated features such as semantic conflict detection. The implementation is careful to avoid busy-waiting or polling when a transaction is blocked, delivering good performance despite contention and high multithreading levels. CCSTM demonstrates that a library-based STM can be usable and performant.

A. Code

Source code for CCSTM is available under a BSD license from http://github.com/nbronson/ccstm .

Acknowledgments

The authors would like to thank Daniel Spiewak and Peter Veentjer for their helpful feedback during the design phase of CCSTM.

This work was supported by the Stanford Pervasive Parallelism Lab, by Dept. of the Army, AHPCRC W911NF-07-2-0027-1, and by the National Science Foundation under grant CNS–0720905.

References

[1] N. G. Bronson, C. Kozyrakis, and K. Olukotun. Feedback-directed barrier optimization in a strongly isolated stm. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–225, New York, NY, USA, 2009. ACM.

[2] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, March 2006.

[3] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, New York, NY, USA, 2009. ACM.

[4] I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47, New York, NY, USA, 2009. ACM.

[5] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[6] T. Harris. Exceptions and side-effects in atomic blocks. In *2004 PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[7] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, July 2005. ACM Press.

[9] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2003.

[11] R. Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. ACM New York, NY, USA, 2008.

[12] G. Korland, N. Shavit, and P. Felber. Noninvasive Java concurrency with Deuce STM (poster). In *SYSTOR '09: The Israeli Experimental Systems Conference*, may 2009. Further details at http://www.deucestm.org/.

[13] V. Menon, S. Balensieger, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *SPAA '08: Proceedings of the 20th ACM Symposium on Parallel Algorithms and Architectures*, 2008.

[14] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 51–62, New York, NY, USA, 2008. ACM.

[15] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *In Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, pages 284–298, 2006.

[16] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, October 2008. ACM.

[17] D. Spiewak. scala-stm. http://github.com/djspiewak/scala-stm.

[18] P. Veentjer and A. Philips. Multiverse. http://multiverse.codehaus.org.