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-only 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. This frees us from the semantic tar pits that surround privatization, strong isolation, and irrevocable system calls. It also allows us to express the STM using Scala classes and methods, a design choice that has far-reaching consequences.

Transactional accesses in CCSTM are performed through instances that implement a trait *Ref*. These transactional references may be long-lived, or may be transient accessors to bulk transactional data such as an array. The syntax for dereferencing *Ref* instances is a pain point for the library-based approach, but the reference-based interface also provides benefits. *Ref* serves as a first-class representation of a transactionally-managed memory location, providing a natural way to express additional STM features such as conditional waiting, non-transactional compare-and-swap, manually-validated reads, and deferrable transformation using pure functions.

In an additional departure from typical STM designs, CCSTM passes the current transactional context through an implicit parameter, rather than dynamically binding the transaction to the current thread. This static transaction scoping allows CCSTM to compete in performance with an STM that performs bytecode rewriting, but it hinders composability. We sketch a potential solution to this problem that combines static scoping for barriers and dynamic scoping for nested transactions.

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 from and stores to shared mutable state, transactional accesses in CCSTM are explicit method calls to a Scala `trait`[1]. We refer to the resulting STM as 'reference-based', because all memory locations managed by the STM are boxed inside transactional references. Both transactional and non-transactional access to the managed references goes through methods implemented by instances of this *Ref*`[A]`.

While a reference-based STM imposes a syntactic burden for simple loads and stores, it also provides advantages. Encapsulating transactionally-managed data eliminates the weak isolation issues that plague transparent STMs. The *Ref* provides a first-class object that names a memory location, which enables the expression of higher-level constructs such as a simple but effective form of semantic conflict detection. The reference also provides a convenient namespace for additional STM functionality.

CCSTM's second departure from the typical STM interface is that it binds the transaction scope statically, rather than dynamically. Transactional access methods in *Ref* take an implicit parameter of type *Txn*, and perform their accesses in the context of that transaction. A useful parallel can be made between Haskell's STM [6] and CCSTM. Haskell's *TVar* corresponds to instances of type *Ref*. *Txn* is not a monad, but it proliferates through STM-enabled methods in exactly the same way as the *STM* monad.

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 introduce `unrecordedRead`, a new STM primitive that relaxes read atomicity while allowing manual validation, and

[1] Custom accessor methods may used to eliminate the syntactic overhead for some situations.

`map`, a simple and safe way of expressing semantic conflict detection. A transaction that reads a reference `x` via `x.map(f)` does not need to roll back if `x` is changed concurrently but `f(x.get)` remains the same (Sections 3.4 and 3.6).

3. We briefly overview CCSTM's implementation, including a novel optimization to reduce the global time-stamp contention of non-transactional isolated writes (Section 4).
4. 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 burdensome, and hypothesize about ways to address the latter (Section 5).
5. We compare the performance of CCSTM to Deuce STM, an STM for the JVM that use bytecode rewriting [10]. We find that CCSTM's implementation as an unprivileged library does not impose a significant performance penalty (Section 6).

2. Background

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 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 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 lazily, but the required engineering effort to add this support to a production quality 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 [16] and Deuce STM [10] STMs for the Java language. While this cost may eventually be considered acceptable, it places a high hurdle to integration into Scala's standard library. An additional drawback of an instrumentation approach is that it is not composable. If module *A* is constructed with the STM *S* and module *B* is constructed with the STM *T*, then *A* and *B* can't 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-only STM tractable. Operator overloading makes transactional loads and stores more concise, and implicit parameters allow the current transaction context to be threaded through the code without explicitly including it in each call. The resulting STM can be considered to be an embedded domain-specific language (DSL) for expressing optimistic concurrency.

2.1 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 manages reads and writes by redirecting them to *barriers*, code fragments that actually implement the atomicity and isolation. These properties can only be guaranteed, however, if no non-transactional code bypasses the barriers and accesses a memory location directly.

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

- The system 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 [14] or a substantial warmup period [1].
- The system 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 [11], but the resulting systems still require whole-program reasoning to guarantee correctness. The privatization problem and its dual, the publication problem, refer to isolation failure for specific useful idioms.
- The system can use types to disallow direct access to any memory location that might be touched transactionally [12]. This can take the form of extending the types and access rules on normal mutable memory locations, or of boxing all transactionally-managed data inside some sort of cell, as in Haskell [6] or Clojure [9]. 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 library-based approach seems the most practical.

2.2 Irrevocable actions and structural conflicts

One of the side effects of an 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. The semantic problems come from actions that the STM cannot isolate or undo, such as I/O or calls to external libraries. In the absence of any additional information about potential conflicts, the only way to execute safely is to serialize. 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 the size over multiple memory locations, no concurrency will actually be available.

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

  def balance: Money = _balance

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

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

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

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

2.3 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 [13] algorithms demonstrated how to use a global time-stamp to efficiently validate a transaction after each read, guaranteeing consistency for all intermediate states. This correctness property was later formalized as *opacity* [4]. CCSTM uses the SwissTM [3] algorithm, which guarantees opacity.

3. CCSTM's interface

As a recurring example of the ways in which CCSTM allows optimistic concurrency to be expressed, consider a class that encapsulates the balance of a checking account[2]. Absent any concurrency control, we might write the code in Figure 1. (Here *Money* is an immutable numeric type suitable for representing quantities of a currency.) 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 gives the transactional code.

[2] This example is from the bank benchmark included in Deuce STM [10].

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

  def balance: Source[Money] = _balance

  def deposit(amount: Money
       )(implicit txn: Txn): Unit = {
    assert(amount >= 0)
    _balance := !_balance + amount
  }

  def withdraw(amount: Money
       )(implicit txn: Txn): Unit = {
    assert(amount >= 0)
    if (!_balance < amount)
      throw new OverdraftException
    _balance := !_balance - amount
  }
}

object Account {
  def transfer(src: Account, dst: Account,
       amount: Money): Unit = {
    new Atomic { def body {
      src.withdraw(amount)
      dst.deposit(amount)
    }}.run()
  }
}
```

Figure 2. One way to implement the atomic balance transfer function using CCSTM. This code uses `unary_!` and `:=` operators (ML-inspired) for performing transactional reads and writes, and expresses the atomic block by extending `Atomic`.

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 ML-inspired `unary_!` and `:=` operators. Section 5.1 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 `extends` 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 [15]. 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 is visually differentiated by including the token `nonTxn`. In Section 3.4 we will introduce `unrecordedRead`, a way of relaxing

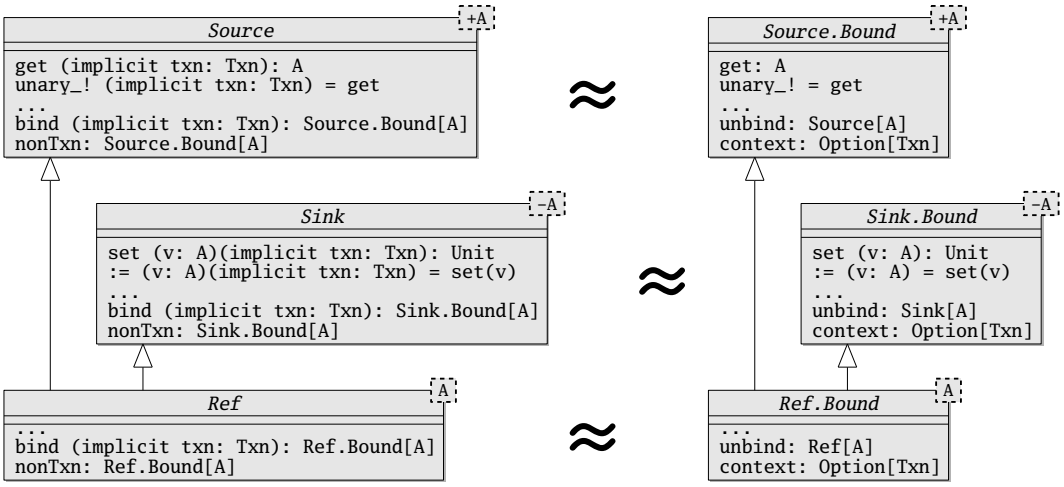

Figure 3. Traits that provide access to an STM-managed memory location. Transactional access can occur through either *Ref* or a *Ref.Bound* returned from *Ref*.bind, non-transactional access occurs through a *Ref.Bound* returned from *Ref*.nonTxn. *Source*[+A] and *Sink*[-A] decompose the covariant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

isolation for reads while retaining the ability to manually validate it.

3.2 Declaring and executing an atomic block

CCSTM currently provides two syntaxes for declaring atomic blocks. The first, shown in Figure 2, involves extending the abstract class *Atomic* and implementing the method body: *Unit*. The base class introduces an implicit *Txn* into the static scope of the body, resulting in a relatively concise syntax for inline transactions. This form produces an extra set of curly braces and several extra tokens, but it uses only one line to open the scope and one to close it. *AtomicFunc*[Z] provides a similar syntax for atomic blocks that return a value.

The second syntax is to pass a function *Txn* => Z to the atomic method on the *STM* object. This syntax is more concise when invoking a method. The *Account*.transfer method might be decomposed into:

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

One of the new features of Scala 2.8 is support for an implicit modifier on parameters to anonymous methods (ScalaTrac ticket #1492). This will make *STM*.atomic more concise for inline transactions, as in (with the addition of an import):

```
atomic { implicit txn: Txn =>
  src.withdraw(amount)
  dst.deposit(amount)
}
```

This will also allow a syntax that leverages for comprehensions. This syntax has a special elegance; it implies that the block may be executed multiple times, which is exactly the essence of optimistic concurrency!

```
for (implicit txn <- atomic) {
  src.withdraw(amount)
  dst.deposit(amount)
}
```

Both of these syntaxes can be supported by an *atomic* object:

```
object atomic {
  def apply[Z](body: Txn => Z): Z
  def foreach[Z](body: Txn => Z): Z
}
```

The decision to statically scope CCSTM's transactions 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-only 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.

3.3 Conditional retry

CCSTM supports the retry and orElse primitives introduced by Harris et al. in Haskell's STM [7], although the current lack of nesting support 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.

Atomic provides a retry method. As a contrived example, the bank could use retry if they wished to delay a withdrawal rather than trigger an OverdraftException:

```
object atomic {
  def withdrawWhenPossible(amount: Money
        )(implicit txn: Txn): Unit = {
```

```
    assert(amount >= 0)
    if (!_balance < amount)
      retry
    _balance := !_balance - amount
  }
```

A chain of *Atomic* or *AtomicFunc* instances may be chained with orElse, a single run() terminates the alternatives. If a collection of *Txn* => **Z** is available then they can be passed to:

```
object STM {
  def retryOrElse[Z](blocks: (Txn => Z)*): Z
}
```

3.4 Relaxed isolation

Some algorithms can benefit from transactional reads that are not atomic, but that 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, or 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.

Previous TM systems have provided several mechanisms for relaxing atomicity and isolation. Open nested transactions allow the actions of a nested transaction to be committed in a non-nested fashion. Escape actions suspend the current transaction temporarily [5]. Early release allows reads to be removed from the read set prior to commit [8]. CCSTM's static transaction scoping makes escape actions trivial. Accesses via nonTxn are escaped when executed by the code that implements an atomic block[3] CCSTM also provides principled support for early release, via *Source.Bound.releasableRead*. This method returns a *ReleasableRead* instance that provides both the read value and a method that removes 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 provide correctness.

As an alternative to a releasable read, CCSTM provides 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 provides 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[4].

3.5 Additional reference operations

What follows is the complete listing of the access operations provided by *Ref.Bound*. Many of these methods have equivalents in *Ref*, although to reduce the API's surface area some are not mirrored.

Source.Bound methods:

- unary_!: T – This is equivalent to a get.
- get : T – Performs a transactional read of the value managed by the bound Ref. If the bounded view was created by a call to nonTxn, the read will be strongly atomic and isolated with respect to all transactions.
- 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 pred is true, in a manner consistent with current context (transactional vs. non-transactional).
- unrecordedRead: UnrecordedRead[T] – Returns an UnrecordedRead instance that wraps the value that would be returned by get. The read will not be added to the transaction's read set (transactional context).
- releasableRead: ReleasableRead[T] – Returns a ReleasableRead instance that wraps the value that would be returned by get. The read will be added to the transaction's read set (transactional context) but it may be removed by a call to ReleasableRead.release(

Sink.Bound methods:

- :=(v: T) – This is equivalent to a set.
- set(v: T) – Updates the value referred to by the bounded Ref. If the bounded view was created by a callto nonTxn, the value will be made available immediately (with an appropriate happens-before relationship).
- tryWrite(v: T): Boolean – Updates the element held by the bound Ref without blocking and returns true, or does nothing and returns false.

Ref.Bound methods:

- readForWrite: T – Returns the same value as that returned by get, but adds the Ref to the write set of the bound transaction context, if any.
- getAndSet(v: T): T – Works like set, but returns the old value.
- compareAndSet(b: T, v: T): Boolean – Equivalent to atomically executing:

```
if (b == get) {
  set(v)
  true
} else
    false
```

- compareAndSetIdentity(b: T, v: T): Boolean – Equivalent to atomically executing:

```
if (b eq get) {
  set(v)
  true
} else
    false
```

- weakCompareAndSet(b: T, v: T): Boolean – Works like compareAndSet , but allows spurious failures.
- weakCompareAndSetIdentity(b: T, v: T): Boolean – Works like compareAndSetIdentity, but allows spurious failures.
- transform(f: T => T) – Atomically replaces the value v stored in the Ref with f(v).
- getAndTransform(f: T => T): T – Atomically replaces the value v stored in the Ref with f(v), returning the old value.

[3] One of the motivations for the creation of CCSTM was to study transactional collection class implementations that interleave escaped actions internally.

[4] 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.

- `tryTransform(f: T => T): Boolean` – Either atomically transforms this reference without blocking and returns true, or returns false.
- `transformIfDef(pf: PartialFunction[T,T]): Boolean` – Atomically replaces the value `v` stored in the `Ref` with `f(v)`, if `pf.isDefinedAt(v)` returning true, otherwise leaves the element unchanged and returns false.

3.6 Semantic conflict detection for reads

Unrecorded reads can be paired with life cycle callbacks to implement a simple yet powerful form of semantic conflict detection. CCSTM's *Txn* allows the user to register a callback that will be invoked whenever the transaction's read set is validated. If an unrecorded read is coupled with a callback that re-reads the value and performs a semantic validation, rollback may be avoided. We use this technique to implement a new abstraction:

```
trait Source[+A] {
  def map[B](f: A => B)(implicit txn: Txn): B
}
```

`x.map(f)` returns the same value as `f(x.get)`, but no rollback is triggered if the post-application value 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 definition of `f`.

Consider a branch that should be taken only if the transactional value of `x` is greater than 100, and assume that a concurrent transaction commits a change of `x` from 200 to 201:

```
if (x.get > 100) { ... }
```

When the comparison is performed directly by the caller, the STM must trigger an optimistic rollback for any concurrent write to `x`, even though the outcome of the conditional check is not affected. If the comparison is moved inside `map` then no rollback is required:

```
if (x.map(_ > 100)) { ... }
```

By allowing the programmer to express more of his or her intention to CCSTM, `map` can reduce rollbacks, leading to better performance and scalability.

4. CCSTM's implementation

CCSTM's implementation is modeled on SwissTM [3]. Version management is lazy, but write permission is acquired eagerly. Time-stamps are allocated 51 bits, making CCSTM immune from counter overflow in all but the most demanding production environments.

4.1 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]]. In some cases these storage optimizations return CCSTM to par with a VM-integrated STM, in some cases they make CCSTM's representation more compact.

4.2 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.

4.3 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 manage 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.

4.4 Polite blocking

An important design goal for CCSTM is support for incremental use inside a larger application. This means that exponential backoff is not a suitable mechanism for blocking. Many STMs target parallel speedups for CPU-bound applications. Assumed (often implicitly) is that all threads belong to the STM and that the number of software threads can be chosen to match the number of hardware execution contexts. CCSTM makes neither of these assumptions, and so takes 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 commit 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 (compare to Java's *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 may be required.

4.5 JVM versus CLR

Scala is designed to target both the JVM and the CLR virtual machines. In its current implementation, CCSTM uses the following classes from *java.util.concurrent* to implement atomic compare-and-swap for fields and array elements: *AtomicInteger*, *AtomicLong*, *AtomicLongArray*, *AtomicReferenceArray*, *AtomicLongFieldUpdater*, and *AtomicReferenceFieldUpdater*. The authors are not experts in 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*.

5. Discussion

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

5.1 Read barrier syntax

The choice of `!x` as the preferred encoding for a transactional read of `x` was inspired by ML's reference syntax. Three alternatives were considered for a concise transactional read:

1. `apply()` – This chains more easily than a prefix operator, without triggering Scala's line merging heuristic. Idiomatic Scala uses `()` to denote a side-effecting method, however, which is confusing.
2. An implicit conversion from *Ref*`[A]` to `A` – This is the most concise, but it interferes with further implicit conversions and can lead to surprises.
3. `unary_!` – This adds only a single character, but it can be confusing when a transactional read is used as part of a boolean condition. It can be difficult to chain without parentheses.

In the code that we have written so far we have mixed `unary_!` and `.get`, falling back on the more verbose form inside conditionals and complicated expressions. For fields that are almost always accessed inside an atomic block, another scheme that works well is to create transactional accessor methods:

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

5.2 Nested transactions

CCSTM currently does not support nested atomic blocks, an important omission. A larger problem, though, is that the static scoping of transactions would make composing such blocks difficult. If a method `m` needs a transaction internally, should it add an `implicit` *Txn* parameter? If it doesn't, then it cannot be composed. If it does, then the caller must always provide an atomic block. If the programmer wishes to provide both a simple and a composable version of the method then he or she must come up with two names.

The need for two versions of `m` parallels the transactional and non-transactional access to a *Ref*, which was resolved there by the *Ref* $\leftrightarrow$ *Ref.Bound* correspondence. Rather than coming up with two method names, the same method name can be used in two classes. *Ref*`.nonTxn` and *Ref.Bound*`.unbind` convert from an instance suitable for one context to the other. While it may be tolerable to manage these parallel classes in the library itself, it seems burdensome to ask the programmer to perform a similar task.

A possible solution is to provide dynamic scoping for the declaration of atomic blocks, but static scoping for barriers. Because nested transactions are likely to be less common than barriers, there is less performance motivation for avoiding the thread-local lookup.

It seems dangerous to mix static and dynamic scoping, but we notice that for all but the most subtle uses, the static and dynamic scopes should be identical. This means that we can provide an execution mode that checks the static scopes against the dynamic ones, and triggers an exception if they don't match. This checking mode would be slower on a per-barrier basis, so like asserts it might be disabled during production use.

For applications that don't spend a large portion of their time in barriers, we should also consider providing full dynamic scoping for transactions. Interestingly, this could be enabled in a per-module fashion by providing an implicit function that performs the required thread-local lookup. The result might look like:

```
class Account ... {
  import STM.dynamic

  def deposit(amount: Money): Unit = {
    assert(amount >= 0)
    _balance := !_balance + amount
  }
}
```

If `deposit` were executed outside a transaction a runtime error would be generated, rather than the compile-time error produced by the existing CCSTM.

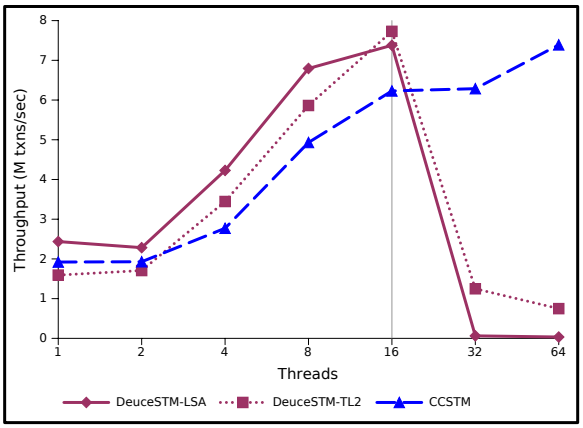

Figure 4. Throughput for the bank benchmark in a low contention scenario, on a machine with 16 hardware thread contexts. The number of accounts is 64 times the number of threads.

6. Performance

To verify that CCSTM's library-based design does not impose a prohibitive performance penalty, we compared it to Deuce STM,

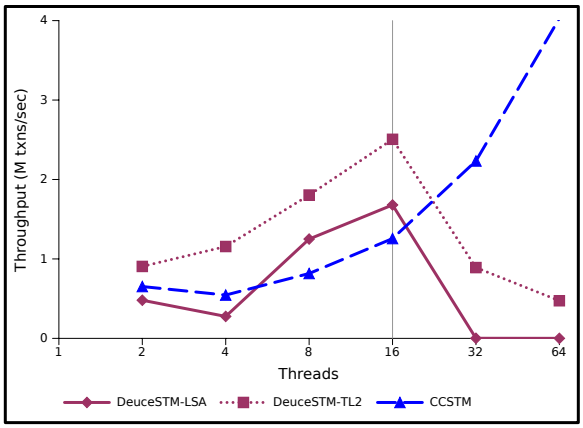

Figure 5. Throughput for a high contention scenario. The number of accounts is equal to the number of threads. Each transaction touches two accounts.

an STM for the JVM that performs bytecode rewriting during class loading [10].

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 2.7.7. We ran our experiments in Sun's Java SE Runtime Environment, build 1.6.0_16-b01, using the HotSpot 64-Bit Server VM. Deuce STM was version 1.2.0.

We performed a direct encoding of Deuce STM's bank benchmark into Scala+CCSTM, and compared this version to the Java original running under both the TL2 and LSA variants of Deuce STM. (We were not able to use Deuce STM to execute Scala code transactionally, due to a verifier error in the instrumented classes.) This benchmark includes its own harness, which we configured so that no overdrafts were triggered. We used a 4 second warmup, and then measured the number of transactions committed during 10 seconds, averaging across three invocations of the JVM. For the low-contention experiment (Figure 4) we set the number of accounts to 64 times the number of threads. For the high-contention experiment (Figure 5) 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, the high-contention runs have fewer conflicts at 32 and 64 threads than for lower thread counts.

The TL2 version of Deuce STM is faster than CCSTM for both experiments when the multi-threading level is less than or equal to one. The difference is largest for the high-contention scenario, where threads are often obstructed. Deuce STM does not block threads that cannot proceed, and it does not even use sleeps or yields. This strategy works well when every thread gets its own hardware context, but results in a catastrophic performance dropoff for higher thread counts. CCSTM's synchronization implementation is more expensive, but yields stable performance even at high multi-threading levels.

Deuce STM and CCSTM have different algorithms and engineering tradeoffs, so these experiments do not allow us to exactly measure the overhead imposed by the library-only design. They do demonstrate, however, that any overhead that does exist is small enough to be tolerable.

7. Conclusion and future direction

CCSTM demonstrates that it is possible to provide a library-only implementation of STM for Scala. The resulting syntax is relatively concise and the performance is on par with a bytecode rewriting STM. The transactional references that encode CCSTM's interface 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.

We found that the primary challenge in our approach was that statically scoped transactions reduce composability. The next step for CCSTM is to explore a hybrid approach that uses static scopes for barriers but dynamic scopes to coordinate nested transactions.

A. Code

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

Acknowledgments

The authors would like to specifically 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] 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.

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

[6] 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.

[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] 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.

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

[10] 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/`.

[11] 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.

[12] 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.

[13] 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 (DISCâĂŹ06*, pages 284–298, 2006.

[14] 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.

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

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