CCSTM: A Library-Based Software Transactional Memory for Scala

Pervasive Parallelism (PPL) Lab. (http://ppl.stanford.edu)

Nathan Bronson, Hassan Chafi, Kunle Olukotun

Can STM Be "Pay For What You Use"?

The Problem

- STM is **one** tool in the parallel programmer's toolbox
- How do we deliver incremental benefit with low risk and incremental cost?

Our Solution

- CCSTM is an unprivileged library
- CCSTM manages only boxed data
  - `Ref[T]` holds a single value
  - `TArray[T]` holds multiple values
- Barriers are explicit method calls
- Scala's good DSL support lets it look nice

Composability of STMs

None – Status Quo

- Entire program can only use a single STM
- STM is risky for libraries, because it adds a whole-system constraint

STM via compiler or runtime instrumentation

Coexistence – Non-concurrent Use

- Components may use different STMs
- Transactions from separate STMs are rarely simultaneously active

STMs that waste thread resources while waiting

Coexistence – Concurrent Use

- Components may use different STMs
- Transactions from separate STMs are not simultaneously active on a thread

CCSTM

Library-based STMs that block using OS locks, or separate instrumentation with static txn scopes

Nesting – Arbitrary Composition

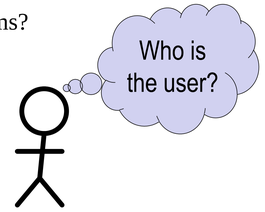
- Arbitrary composition

2-Phase commit with a standardized API?

Syntax Comparison

	Deeply-Integrated	CCSTM
Mutable shared state	`var x = ☐`	`val x = Ref(☐)`
Read	`☐ = x`	`☐ = x()`
Write	`x = ☐`	`x() = ☐`
Atomic block	`atomic { ☐ } orElse { ☐ }`	`atomic { implicit t => ☐ } orAtomic { implicit t => ☐ }`

- Automatic generation of lock-free algorithms?
- An easier replacement for locks?
- A safer replacement for locks?
- A scalable replacement for locks?
- Speculative parallelism?
- **A way to compose atomic operations**



- Collection class implementers?
- Sequential programmers?
- **Parallel programmers**

Report Cards: Deeply-Integrated Vs. Library-Based

Student: Deeply-Integrated STM

Subject: End User Experience

Ease of use + Simple mental model … – … so long as you avoid I/O (hard to roll back)	A-
Composability of code using transactions + Nesting has expected semantics, no deadlocks	A
Testability + Invariants are preserved throughout a transaction, even if other code doesn't synchronize properly	A+
Performance – Single-thread overheads are higher than locks	B
Scalability + Reads often scale better than locks + Writes often scale like careful fine-grained locking	A

Subject: Language Designer's Perspective

Ease of language integration – Strong atomicity and isolation require extensions to the type system	needs improvement
Composability of implementations – Only one STM can be used in a VM	
Testability – Tight integration requires a large up-front design before users can provide feedback	
Performance – Code that doesn't use transactions may have reduced performance, especially during startup	
Scalability – If any part of a system uses STM, all of the classes must be instrumented	

Student: Unprivileged Library STM

Subject: End User Experience

Ease of use + Clean and concise for new code – Existing code must be modified	B+
Composability + Just as good as deeply-integrated STM	A
Testability + Local reasoning still possible – No checking that shared mutable state is in **Ref**	A
Performance – Still has a single-thread performance penalty + Single-operation transactions are optimized	B+
Scalability + Easier to provide advanced conflict-avoidance strategies	A+

Subject: Language Designer's Perspective

Ease of language integration + None needed	★
Composability of implementations + Coexistence of STMs is fine – Atomic blocks from different STMs don't nest	★
Testability + CCSTM can be used independently	★
Performance + Components only pay for what they use	★
Scalability + Only components using CCSTM are aware of it	★

An Example

```
class IntSet {
  private class Node(val e: Int, next0: Node) {
    val next = Ref(next0)
  }
  private val header = new Node(-1, null)

  def contains(e: Int) = atomic { implicit t =>
    @tailrec def loop(cur: Node): Boolean = {
      cur != null && (cur.e == e || loop(cur.next()))
    }
    loop(header.next())
  }
  def await(e: Int): Unit = atomic { implicit t =>
    if (!contains(e)) retry
  }
  def add(e: Int): Unit = atomic { implicit t =>
    @tailrec def loop(prev: Node) {
      val cur = prev.next()
      if (cur == null || cur.e > e)
        prev.next() = new Node(e, cur)
      else if (cur.e != e)
        loop(cur)
    }
    loop(header)
  }
  def addAll(ee: Int*): Unit = atomic { implicit t =>
    for (e <- ee) add(e)
  }
}
```

API Subset

```
class Source[+T] {
  // Performs a transactional read and checks that it is consistent with
  // all reads already made by txn.
  def get(implicit txn: Txn): T
  def apply()(implicit txn: Txn) = get

  // Returns a view that performs single-operation transactions.
  def single: Source.View[T]

  // Returns f(get), possibly reevaluating f to avoid rollback if a
  // conflicting change is made but the old and new values are equal
  // after application of f.
  def getWith[Z](f: T => Z)(implicit txn: Txn): Z
  ...
}
```

```
class Sink[-T] {
  // Performs a transactional write.
  def set(v: T)(implicit txn: Txn): T
  def update(v: T)(implicit txn: Txn) = set(v)

  // Returns a view that performs single-operation transactions.
  def single: Sink.View[T]
  ...
}
```

```
object atomic {
  // Executes block in a transaction.
  def apply[Z](block: Txn => Z): Z
}
```

```
class Ref[T] extends Source[T] with Sink[T] {
  // Performs a transactional write, returning the old value.
  def swap(v: T)(implicit txn: Txn): T

  // Atomically sets the value to f(get), possibly (re)evaluating f
  // later in the transaction to avoid rollback.
  def transform(f: T => T)(implicit txn: Txn)

  // If pf.isDefined(get), transforms by pf and returns true,
  // otherwise returns false.
  def transformIfDefined(pf: PartialFunction[T,T]
      )(implicit txn: Txn): Boolean

  // If numeric operations are available for T, increments by d.
  def +=(d: T)(implicit t: Txn, ops: Numeric[T])

  // Returns a view that performs single-operation transactions.
  def single: Ref.View[T]
  ...
}
```

Interface Innovations

Hybrid Transaction Scoping

- Static scoping for barriers
  - Better compile-time safety
  - Avoids `ThreadLocal` overheads
  - `Txn` is passed to barrier via a Scala `implicit`
- Dynamic scoping for atomic blocks
  - Retains full composability for transactional code
  - Reintroduce static scope with a nested atomic

Single-Operation Transactions

- Atomic if outside a transaction, nested if inside
  `atomic { implicit t => ref.op }` becomes `ref.single.op`
- Strongly atomic reads and writes
- Optimized implementations for read-then-write
  - `x.single.transform(f), swap, compareAndSet, transformIfDefined, …`

Unrecorded Reads

```
class UnrecordedRead[T] {
  def value: T ; def stillValid: Boolean }
```

- Metadata is captured to allow manual validation
- Combine with lifecycle callbacks to implement custom validation

Read Via a Pure Function

- `x.getWith(f)` returns `f(x())`
  Example: `if (size.getWith(_ == 0)) empty…`
- No rollback unless result changes
- Restricted form of Abstract Nested Transaction

Algorithmic Details

SwissTM with TL2's GV6 Global Clock

- Lazy versioning (write buffer)
- Eager detection of write-write conflicts
- Opacity guaranteed

CCSTM-Specific

- Extensible mapping between data and metadata
- Contention management and `retry`/`orAtomic` use JVM monitors and `wait`/`notifyAll`
- Global clock relaxation for non-transactional writes
- *To come: Partial rollback of associative write buffer*

Contact information

Project Hosting (BSD License)

`http://ppl.stanford.edu/ccstm`

E-mail Addresses

`{nbronson, hchafi, kunle}@stanford.edu`