CCSTM: A Library-Based Software Transactional Memory for Scala

Nathan Bronson, Hassan Chafi and Kunle Olukotun

Stanford University

ScalaDays 2010
The Context

How do threads coordinate their access to **shared mutable state**?

1: Don’t do it?
2: Locks?

**Solution should be:**
- Easy to use
- Composable
- Testable
- Performant
- Scalable
Software Transactional Memory*

Atomic execution of multiple loads and stores
- Declarative syntax
- Accesses needn’t be known ahead of time
- Parallel execution whenever possible

```
// Thread A - push x
atomic begin
  val n ← new Node(x)
  n.next ← head
  head ← n
end

// Thread B - push y
atomic begin
  val n ← new Node(y)
  n.next ← head
  head ← n
end
```

* - The ideal
In concurrent programming, an operation is linearizable, *atomic*, indivisible or uninterruptible if it appears to take effect instantaneously.
So Far

Atomic blocks are like a magic replacement for locks

- No serialization on coarse-grained locks
- No complicated fine-grained locking schemes
- No worrying about deadlock
Q: How can TM execute atomic blocks in parallel if their read and write sets are not known in advance?

A: Speculatively, fixing with rollback+retry

// Thread A
atomic begin
  ... // lots of work
  x = 1
end

// Thread B
atomic begin
  ... // lots of work
  x = 2
undo stores, retry txn
atomic begin
  ... // lots of work
  x = 2
end
Supporting Speculative Execution

Transactional reads
- Loads must be remembered, to check for conflicts

Transactional writes
- Both original and speculatively-modified versions of data must be retained
  - *Undo log: original version on the side*
  - *Write buffer: speculative version on the side*

Control flow
- Non-local control transfer is possible from any memory access to the beginning of the transaction
# Ideal STM (Graded by the User)

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of use</td>
<td>Simple mental model …</td>
<td>A-</td>
</tr>
<tr>
<td></td>
<td>… so long as you avoid I/O (hard to roll back)</td>
<td></td>
</tr>
<tr>
<td>Composability of code using transactions</td>
<td>Nesting has expected semantics, no deadlocks</td>
<td>A</td>
</tr>
<tr>
<td>Testability</td>
<td>Invariants are preserved throughout a transaction, even if other code doesn’t synchronize properly</td>
<td>A+</td>
</tr>
<tr>
<td>Performance</td>
<td>Single-thread overheads are higher than locks</td>
<td>B</td>
</tr>
<tr>
<td>Scalability</td>
<td>Reads often scale better than locks</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>Writes often scale like the best fine-grained locking</td>
<td></td>
</tr>
</tbody>
</table>
Compiling an Atomic Block for STM

atomic begin

val n ← new Node(x)

n.next ← head

head ← n

end

val txn = new Txn()
don {try {
    txn.begin()

    val n = new Node(x)(txn)

    val tmp = txn.readAnyRef[Node](this, HeadOffset)
    txn.write(n, NextOffset, tmp)
    txn.write(this, HeadOffset, n)
} catch {
    case RollbackError => {} 
    case ex => txn.userException(ex)
}
} while (!txn.attemptCommit())
Who Instruments the Code?

- Scala source
- Class files
- Loaded bytecode
- Machine code

Scalac or plugin?

Bytecode rewriting?

VM JIT?
How Do We Compile Atomic Blocks?

Loads and stores inside *atomic* are redirected to STM

“Inside” is a dynamic scope

Two copies of every method are needed
How Do We Compile Atomic Blocks?

STM creates illusion of atomicity and isolation

Type system extended to segregatetxn and non-txn data

Too slow to send all non-txn accesses to STM

or

User error ➔ loss of atomicity, values from thin air, “catch fire”
Ideal STM (Graded by Martin)

Ease of language integration
- Strong atomicity and isolation require extensions to the type system

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

needs improvement
Can We Pass Both Classes?

Transactional memory is a nice abstraction for the user.

Can we provide most of the benefit without intrusive language modifications?
CCSTM: Library-Based STM

No instrumentation, so STM must be called explicitly
Managed data encapsulated by \texttt{Ref[A]}

<table>
<thead>
<tr>
<th>Deeply-Integrated</th>
<th>CCSTM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mutable shared state</strong></td>
<td>\texttt{val x = Ref(x)}</td>
</tr>
<tr>
<td><strong>Read</strong></td>
<td>\texttt{x = x()}</td>
</tr>
<tr>
<td><strong>Write</strong></td>
<td>\texttt{x := x}</td>
</tr>
<tr>
<td><strong>Atomic block</strong></td>
<td>\texttt{STM.atomic { implicit t =&gt; x } }</td>
</tr>
</tbody>
</table>
trait Ref[A] – Implementations

Decomposed into `Source[+A]` and `Sink[-A]`
- From Daniel Spiewak’s Scala STM

Storage Ref-s store a mutable value directly
- `TBooleanRef`, `TByteRef`, … `TAnyRef[A]`
- `object Ref`’s `apply(v)` picks the right implementation
- Internal representation is flexible
  - `TPairRef[A,B]` *deconstructs and reconstructs its value*
  - `StripedIntRef`, `LazyConflictIntRef` *reduce conflicts*

Proxy Ref-s are constructed on demand
- `TArray[A]` avoids long-term boxing
- `TxnFieldUpdater` instances create Ref-s for any property with volatile semantics
trait Ref[A] – More Operations

def get: A – non-operator read

def map[Z](f: A => Z): Z – no rollback if f(get) doesn’t change

def unrecordedRead: UnrecordedRead[A] – no conflict checking

def await(pred: A => Boolean) – retriestxn if !pred(get)

def set(v: A) – non-operator write

def transform(f: A => A) – equivalent to set(f(get))

def transformIfDefined(pf: PartialFunction[A, A]): Boolean – generalizes compareAndSet

def tryWrite(v: A): Boolean – fails instead of blocking

def getAndSet(v: A): A – returns the previous value

...
Scoping of the Current Txn

How is the active \texttt{Txn} found by \texttt{Ref}’s methods?

- STM participates in the compilation of all code
  
  Option 1: Add a \texttt{Txn} parameter during translation
  Option 2: Add a \texttt{currentTxn} field to \texttt{Thread}

- Dynamic lookup
  
  Option 3: \texttt{ThreadLocal}
  \textit{ Undesirable performance overhead }

- Static lookup
  
  Option 4: \texttt{Ref}’s methods take an implicit \texttt{Txn}
  \textit{ Hinders composability }
Our Solution: Hybrid Scoping

Dynamic scoping for atomic blocks
- Using ThreadLocal

Static scoping for Ref’s methods
- Using an implicit Txn parameter
  (Omitted from the method list two slides ago)

Don’t have an implicit Txn available?
*Just declare a new atomic block*
- If no txn was active, you probably needed one anyway
- If a txn is in the dynamic scope, the new block nests
Single-Operation Transactions

What happens if a `Ref` method is called outside an atomic block?

1. Compile time error?
   Makes it harder to accidentally omit atomic blocks
2. Execute as if in its own transaction?
   Convenient, especially with `Ref`’s powerful methods
3. Both of the above
   Add an alternate syntax for single-operation txns

`Ref.single` returns a view with methods that mirror `Ref`’s, but that need no implicit `Txn`

```
STM.atomic { implicit t =>
    x := x() + 1
}
```

is equivalent to

```
x.single.transform { _ + 1 }
```
## CCSTM (Graded by the User)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Grade</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ease of use</strong></td>
<td></td>
<td>+ Clean and concise for new code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>− Existing code must be modified</td>
</tr>
<tr>
<td><strong>Composability</strong></td>
<td></td>
<td>+ Just as good as deeply-integrated STM</td>
</tr>
<tr>
<td><strong>Testability</strong></td>
<td></td>
<td>+ Local reasoning still possible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>− No checking that shared mutable state is in Ref</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td></td>
<td>− Still has a single-thread performance penalty</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ Single-operation transactions are optimized</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td></td>
<td>+ Easier to provide advanced conflict-avoidance strategies</td>
</tr>
</tbody>
</table>
# CCSTM (Graded by Martin)

<table>
<thead>
<tr>
<th>Category</th>
<th>Grade</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ease of language integration</td>
<td>★</td>
<td>None needed</td>
</tr>
<tr>
<td>Composability of implementations</td>
<td>★★★</td>
<td>Coexistence of STMs is fine&lt;br&gt;Atomic blocks from different STMs don’t nest</td>
</tr>
<tr>
<td>Testability</td>
<td>★★★</td>
<td>CCSTM can be used independently</td>
</tr>
<tr>
<td>Performance</td>
<td>★★★</td>
<td>Components only pay for what they use</td>
</tr>
<tr>
<td>Scalability</td>
<td>★</td>
<td>Only components using CCSTM are aware of it</td>
</tr>
</tbody>
</table>
Scala Features We Enjoyed

- **Operator overloading** – concise reads and writes
- **Anonymous methods** – concise atomic blocks
- **Type inference** – less clutter when declaring `Ref`-s
- **Mixins** – reduced code duplication
- **Implicit parameters** – improves performance, allows static checking of `Ref` usage
- **Companion object factory methods, class manifests** – storage optimizations for `Ref[A]` and `TArray[A]`
- **Abstract type constructors** – lets `TxnFieldUpdater` handle fields of generic classes
- **JVM integration** – allowed use of advanced features from `java.util.concurrent.atomic`
- **@specialized** – future performance enhancements?
Questions?

http://ppl.stanford.edu/ccstm
Dealing with Shared Mutable State

Solution #1 – Avoid mutable state entirely
Programs are functions from input to output
No variables, just values

Problem: User must (re)create their own abstractions to model identity

Identity: a stable logical entity associated with a series of different values over time*

Dealing with Shared Mutable State

Solution #1 – Avoid mutable state entirely

Solution #2 – Avoid *shared* mutable state
  Use explicit inter-thread (inter-actor) communication
  Mutable state is directly accessed only by its owning context

Problem: Coordination between multiple actors can be complicated

Problem: Best data-to-actor binding might be contrived or dynamic
Dealing with Shared Mutable State

Solution #1 – Avoid mutable state entirely
Solution #2 – Avoid shared mutable state
Solution #3 – Prevent conflicting accesses
  Protect accesses using locks

Problem: Not declarative
  Code shows one synchronization strategy, not a desired property of the program

Problem: Simplicity ↔ scalability tradeoff
  Coarse-grained locks → simple, doesn’t scale
  Fine-grained locks → tricky, might scale

Problem: Not composable
  Correctness is a whole-program property
Dealing with Shared Mutable State

Solution #1 – Avoid mutable state entirely
Solution #2 – Avoid shared mutable state
Solution #3 – Prevent conflicting accesses
Solution #4 – Back up and retry after a conflict
Software transactional memory

```c
// Thread 1
atomic {
  x.bal = x.bal - 20
  y.bal = y.bal + 20
}

// Thread 2
atomic {
  y.bal = y.bal - 20
  x.bal = x.bal + 20
}
```