Transactional Boosting: A Methodology for Highly Concurrent Transactional Objects

Eric Koskinen
Brown University

Joint work with

Maurice Herlihy
Brown University
Everyone loves STM
Everyone loves STM

locking doesn’t scale
Everyone loves STM

- Locking doesn’t scale
- Replace locking with software transactions
Everyone loves STM

- Locking doesn’t scale
- Replace locking with software transactions
Everyone loves STM

- Locking doesn’t scale
- Replace locking with software transactions

But numerous issues remain.
Everyone loves STM

- Locking doesn’t scale
- Replace locking with software transactions

- But numerous issues remain.
- Here’s one ...
Concurrent Skew Heap

```
  4
 / \
9   7
/   /   \
11  75  23
```
Concurrent Skew Heap

Thread A

insert(91)
Concurrent Skew Heap

Thread A

insert(91)
Concurrent Skew Heap

Thread A
insert(91)
Concurrent Skew Heap

Thread A
insert(91)
Concurrent Skew Heap

Thread A
insert(91)
Concurrent Skew Heap

Thread A

insert(91)
Concurrent Skew Heap

Thread A

insert(91)
Concurrent Skew Heap

Thread A

insert(91)
Concurrent Skew Heap

Thread A

insert(91)
Concurrent Skew Heap

Thread A

insert(91)
Concurrent Skew Heap

Thread A

insert(91)
Concurrent Skew Heap

Thread A
insert(91)
Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()
Concurrent Skew Heap

Thread A
insert(91)

Now, in STM: 91

Thread B
removeMin()

91 -> 9
9 -> 4, 7
4 -> 11, 75
11 -> 91
75 -> 7
7 -> 23
Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()

Now, in STM:
91
Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()

Now, in STM:

```
atomic {
    ...
    tmp = root.left;
    root.left := root.right;
    root.right := tmp;
    ...
}
```

```
atomic {
    ...
    a := root.left;
    b := root.right;
    ...
}
```
Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()

Now, in STM:

```
atomic {
  ...  
  tmp = root.left;
  root.left := root.right;
  root.right := tmp;
  ...  
}
```

```
atomic {
  ...  
  a := root.left;
  b := root.right;
  ...  
}
```
Concurrent Skew Heap

Thread A
insert(91)

Thread B
removeMin()

Now, in STM:

```
atomic {
  ... 
  tmp = root.left;
  root.left := root.right;
  root.right := tmp;
  ... 
}
```

false conflict!!!
The Problem

- Highly concurrent objects
- Become sequential in a transaction
The Problem

- Highly concurrent objects
- Become sequential in a transaction
The Problem

- Highly concurrent objects
- Become sequential in a transaction

Cause:
The Problem

- Highly concurrent objects
- Become sequential in a transaction

Cause:
- False conflicts
The Problem

- Highly concurrent objects
- Become sequential in a transaction

Cause:
- False conflicts
- No distinction between thread synchronization and transactional synchronization
Transactional Boosting

- Black-box Linearizable base object
- Given semantics: commutative methods and inverse methods
- Solve synchronization and recovery without tracking read/write sets
- Reuse highly concurrent linearizable objects as highly concurrent transactional objects
Transactional Boosting

- Black-box Linearizable base objects
- Given semantics: commutative and inverse methods
- Solve synchronization and recovery without tracking read/write sets
- Reuse highly concurrent linearizable objects as highly concurrent transactional objects

Highly concurrent transactional objects for free!
Simple Example: A Set

7 2 5 3 (Implemented as a linked list)

Methods

- `add(x)/bool`
- `remove(x)/bool`
- `contains(x)/bool`
Simple Example: A Set

Method

Inverse

7 → 2 → 5 → 3
Simple Example: A Set

Method

Inverse

<table>
<thead>
<tr>
<th>add(x) / false</th>
<th>noop()</th>
</tr>
</thead>
</table>

7 → 2 → 5 → 3
Simple Example: A Set

Method | Inverse
---|---
add(x)/false | noop()
add(x)/true | remove(x)/true

7 → 2 → 5 → 3
Simple Example: A Set

Method | Inverse
---|---
add(x)/false | noop()
add(x)/true | remove(x)/true
remove(x)/false | noop()
Simple Example: A Set

<table>
<thead>
<tr>
<th>Method</th>
<th>Inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(x)/false</code></td>
<td><code>noop()</code></td>
</tr>
<tr>
<td><code>add(x)/true</code></td>
<td><code>remove(x)/true</code></td>
</tr>
<tr>
<td><code>remove(x)/false</code></td>
<td><code>noop()</code></td>
</tr>
<tr>
<td><code>remove(x)/true</code></td>
<td><code>add(x)/true</code></td>
</tr>
</tbody>
</table>
Simple Example: A Set

Method | Inverse
--- | ---
add(x)/false | noop()
add(x)/true | remove(x)/true
remove(x)/false | noop()
remove(x)/true | add(x)/true
contains(x)/_ | noop()
Simple Example: A Set

Commutativity

7 2 5 3
Simple Example: A Set

\[ \text{Commutativity} \]

\[ \text{add}(x) \_ \Leftrightarrow \text{add}(y) \_ \quad x \neq y \]
Simple Example: A Set

Commutativity

<table>
<thead>
<tr>
<th></th>
<th>add(x)/_ ⇔ add(y)/_</th>
<th>remove(x)/_ ⇔ remove(y)/_</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)/_</td>
<td>⇔</td>
<td>remove(y)/_</td>
</tr>
<tr>
<td>x≠y</td>
<td></td>
<td>x≠y</td>
</tr>
</tbody>
</table>
### Simple Example: A Set

#### Commutativity

<table>
<thead>
<tr>
<th>Operation</th>
<th>Commutativity</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)/_</td>
<td>⇔ add(y)/_</td>
<td>x ≠ y</td>
</tr>
<tr>
<td>remove(x)/_</td>
<td>⇔ remove(y)/_</td>
<td>x ≠ y</td>
</tr>
<tr>
<td>contains(x)/_</td>
<td>⇔ contains(y)/_</td>
<td>x ≠ y</td>
</tr>
</tbody>
</table>
Simple Example: A Set

Commutativity

<table>
<thead>
<tr>
<th>Operation</th>
<th>Commutative Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)/_</td>
<td>$\iff$ add(y)/_</td>
</tr>
<tr>
<td>remove(x)/_</td>
<td>$\iff$ remove(y)/_</td>
</tr>
<tr>
<td>contains(x)/_</td>
<td>$\iff$ contains(y)/_</td>
</tr>
</tbody>
</table>

$x \neq y$

all commute with each other
Simple Example: A Set

Commutativity

<table>
<thead>
<tr>
<th>Operation</th>
<th>Commutative Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)/*</td>
<td>add(y)/*</td>
</tr>
<tr>
<td>remove(x)/*</td>
<td>remove(y)/*</td>
</tr>
<tr>
<td>contains(x)/*</td>
<td>contains(y)/*</td>
</tr>
</tbody>
</table>

All commute with each other

7 → 2 → 5 → 3
**Simple Example: A Set**

**Commutativity**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Condition</th>
<th>Commutes when</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(x)/false</td>
<td>x ≠ y</td>
<td>false</td>
</tr>
<tr>
<td>remove(x)/false</td>
<td>x ≠ y</td>
<td>false</td>
</tr>
<tr>
<td>contains(x)/false</td>
<td>x ≠ y</td>
<td>false</td>
</tr>
</tbody>
</table>

All operations commute with each other.
Example 2: Priority Queue

(Implemented as a Skew Heap)
Example 2: Priority Queue

```
Method   | Inverse
--------|--------
removeMin(x) | insert(x)
```

Diagram of a priority queue:
- Root node: 4
- Left child: 9
  - Left child: 11
  - Right child: 75
- Right child: 7
  - Left child: 23
- Leaf node: 91
Example 2: Priority Queue

- **Method**
  - removeMin(0/x)
  - min(0/x)

- **Inverse**
  - insert(x)
  - noop()
Example 2: Priority Queue

Method | Inverse
--- | ---
removeMin∅/x | insert(x)
min∅/x | noop
insert(x) | insertInv(x)
**Example 2: Priority Queue**

<table>
<thead>
<tr>
<th>Method</th>
<th>Inverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeMin(x)</td>
<td>insert(x)</td>
</tr>
<tr>
<td>min(x)</td>
<td>noop()</td>
</tr>
<tr>
<td>insert(x)</td>
<td>insertInv(x)</td>
</tr>
</tbody>
</table>

Diagram:
- Nodes labeled with numbers 9, 7, 4, 11, 75, 23.
- Node 9 is marked with an 'X'.

Table:
- Method: removeMin(x), min(x), insert(x)
- Inverse: insert(x), noop(), insertInv(x)
Example 2: Priority Queue
Example 2: Priority Queue

Commutativity

\[ \text{insert}(x) \Leftrightarrow \text{insert}(y) \]
Example 2: Priority Queue

Commutativity

- $\text{insert}(x) \leftrightarrow \text{insert}(y)$
- $\text{removeMin}(x) / \Rightarrow \text{insert}(y) \quad x \leq y$
Example 2: Priority Queue

Commutativity

- \( \text{insert}(x) \Leftrightarrow \text{insert}(y) \)
- \( \text{removeMin}() / x \Leftrightarrow \text{insert}(y) \quad x \leq y \)
- \( \text{min}() / x \Leftrightarrow \text{min}() / x \)
Outline
Outline

✓ Black-box linearizable base object
Outline

✓ Black-box linearizable base object
✓ Commutativity and inverses
Black-box linearizable base object
Commutativity and inverses

Now, where's my free transactional object?
Outline

✓ Black-box linearizable base object
✓ Commutativity and inverses

Now, where’s my free transactional object?

Synchronization
Outline

✓ Black-box linearizable base object
✓ Commutativity and inverses

Now, where's my **free** transactional object?

Synchronization

Recovery
Txn Synchronization

Thread A

Thread B

Thread C

Black-box Concurrent Data Structure
Txn Synchronization

Thread A

Thread B

Thread C

Black-box Concurrent Data Structure
Txn Synchronization

Thread A

Thread B

Thread C

Black-box Concurrent Data Structure
Txn Synchronization

Thread A

Thread B

Thread C

Black-box Concurrent Data Structure
Transactions synchronize with abstract locks
Transactions synchronize with abstract locks

Base object handles thread synchronization (e.g. lock-free)
Txn Synchronization

- Acquire **abstract locks** before method invocation.

- A lock **conflict** means methods don't commute.

- Thus, abstract locks ensure that only commuting method calls execute concurrently.
Outline

✓ Black-box linearizable base object
✓ Commutativity and inverses

Now, where's my free transactional object?

✓ Synchronization: Abstract Locks

Recovery
Recovery

- Transaction invokes various methods
- Each successful method call, log the inverse method
- On abort replay the log:
  - Invoke inverses
  - Release locks
Outline

✓ Black-box linearizable base object
✓ Commutativity and inverses
Now, where’s my free transactional object?
✓ Synchronization: Abstract locks
✓ Recovery: Log inverses
public class BoostedSet<Integer> {
    ConcurrentSet<Integer> baseSet;
    LockKey abstractLock;
    ...
    public boolean add(int v) {
        abstractLock.lock(v);
        boolean result = baseSet.add(v);
        if ( result ) {
            Thread.onAbort(new Runnable() {
                public void run() { baseSet.remove(v); }
            });
        }
        return result;
    }
}
public class BoostedSet<Integer> {
    ConcurrentSet<Integer> baseSet;
    LockKey abstractLock;
    ...
    public boolean add(int v) {
        abstractLock.lock(v);
        boolean result = baseSet.add(v);
        if ( result ) {
            Thread.onAbort(new Runnable() {
                public void run() { baseSet.remove(v); }
            });
        }
        return result;
    }
}
public class BoostedSet<Integer> {
    ConcurrentSet<Integer> baseSet;
    LockKey abstractLock;
    ...
    public boolean add(int v) {
        abstractLock.lock(v);
        boolean result = baseSet.add(v);
        if (result) {
            Thread.onAbort(new Runnable() {
                public void run() {
                    baseSet.remove(v);
                }
            });
        }
        return result;
    }
}
Boosted Concurrent Set

```java
public class BoostedSet<Integer> {
    ConcurrentSet<Integer> baseSet;
    LockKey abstractLock;
    ...
    public boolean add(int v) {
        abstractLock.lock(v);
        boolean result = baseSet.add(v);
        if (result) {
            Thread.onAbort(new Runnable() {
                public void run() {
                    baseSet.remove(v);
                }
            });
        }
        return result;
    }
}
```
Boosted Concurrent Set

```java
public class BoostedSet<Integer> {
    ConcurrentSet<Integer> baseSet;
    LockKey abstractLock;
    ...
    public boolean add(int v) {
        abstractLock.lock(v);
        boolean result = baseSet.add(v);
        if (result) {
            Thread.onAbort(new Runnable() {
                public void run() {
                    baseSet.remove(v);
                }
            });
        }
        return result;
    }
}
```
Experiments

- Boosting implementation in C
- Based on Transactional Locking 2 (TL2)
- Modified STAMP benchmarks to use Boosting
  - vacation -- A travel reservation system
  - kmeans -- A clustering technique
- Ran on 8-way 2.0 GHz Xeon processor
Experiments

Normalized throughput for two STAMP benchmarks
Summary

- Concurrent objects are sequential in a txn
- Use semantics for synchronization, recovery
- No read/write sets → avoid false conflicts
- Highly concurrent transactional objects ... for free!
Open Nested Txns
Open Nested Txns

ONT addresses same problem by releasing effects of nested txns to parent
Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
- Just a mechanism; no methodology
Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
- Just a mechanism; no methodology
- Might deadlock in abort handler (!)
Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
- Just a mechanism; no methodology
- Might deadlock in abort handler (!)
- Lock-coupling doesn’t correspond to nesting
Open NestedTxns

- ONT addresses same problem by releasing effects of nested txns to parent.
- ONT, Boosting both use abstract locks.
- Just a mechanism; no methodology.
- Might deadlock in abort handler (!).
- Lock-coupling doesn’t correspond to nesting.
- Requires re-implementing base object.
Open Nested Txns

- ONT addresses same problem by releasing effects of nested txns to parent
- ONT, Boosting both use abstract locks
- Just a mechanism; no methodology
- Might deadlock in abort handler (!)
- Lock-coupling doesn’t correspond to nesting
- Requires re-implementing base object
- No r/w sets with Boosting: base object perf
No existing STM mechanism approaches lock-free algorithms

- Dice et al. Transactional Locking II. DISC 2006.

This Paper:

- Proofs in Brown Technical Report CS-07-08
Thanks!

Questions?
Lock Comparison

![Graph showing comparison of Txns/Sec across different threads for 2PLock and KeyLock.]
BOOSTING
BOOSTING

atomic {   }

}
Boosting

\[
\text{atomic } \{ \} \}
\]
Boosting

atomic { Concurrent Object }

semantics!