Open Nesting in Software Transactional Memory

Paper by: Ni et al.
Presented by: Matthew McGill
Transactional Memory

- Transactional Memory is a concurrent programming abstraction intended to be
  - as scalable as fine-grained locking
  - much easier to reason about than fine-grained locking
- Transactions are easy to reason about because they are *serializable*
Nested Transactions

• Transactions should be *composable*

• Clients of `Library` should be able to reason about their own code without being aware of how transactions might be used in `Library`’s code

• We have already discussed several options for composing transactions

```java
void foo() {
  atomic {
    ...
    Library.bar();
    ...
  }
}
```
Nested Transactions

• Closed nesting
  • preserves the semantics of the top-most transaction
  • *transaction flattening*: correct, but performs poorly
  • *partial abort*: empirical results from previous papers that we’ve covered suggest it’s not much better than flattening
Nested Transactions

• Open nesting: “applying conflict detection at a higher level of abstraction”
  • alternately: allowing “smart” programmers to violate physical serializability in order to increase concurrency while manually preserving abstract serializability

• When open nested transactions commit, effects are immediately visible; conflict detection and cleanup must be done by the programmer
Suppose $s$ represents a set, implemented as a linked list:

// thread 1
atomic {
  s.insert(10);
  s.insert(20);
}

// thread 2
atomic {
  s.insert(30);
}

What are some possible values for $s$?
Suppose $s$ represents a set, implemented as a linked list:

```
// thread 1
atomic {
  s.insert(10);
  s.insert(20);
}
```

```
// thread 2
atomic {
  s.insert(30);
}
```

Physical view:
Suppose $s$ represents a set, implemented as a linked list:

```c
// thread 1
atomic {
  s.insert(10);
  s.insert(20);
}
```

```c
// thread 2
atomic {
  s.insert(30);
}
```

Physical view:

```
10 → 20 → 30  30 → 10 → 20  10 → 30 → 20  10 → 30
```

physically serializable
Suppose $s$ represents a **set**, implemented as a linked list:

```
// thread 1
atomic {
  s.insert(x);
  s.insert(y);
}
```

```
// thread 2
atomic {
  s.insert(z);
}
```

**Abstract view:**

\{10, 20, 30\} \quad \{10, 30\}

**Physical view:**

```
10 -> 20 -> 30
```

```
30 -> 10 -> 20
```

```
10 -> 30 -> 20
```

```
10 -> 30
```

**physically serializable**
Suppose $s$ represents a set, implemented as a linked list:

```
// thread 1
atomic {
  s.insert(x);
  s.insert(y);
}
```

```
// thread 2
atomic {
  s.insert(z);
}
```

**Physical view:**

```
{10, 20, 30}
```

```
{10, 30}
```

**Abstract view:**

```
{10, 20, 30}
```

```
{10, 30}
```

Invalid

physically serializable

abstractly serializable
Nested Transactions

• Intuition for Open Nesting
  • Remember “The Problem with Threads”?
  • “Deterministic ends should be accomplished with deterministic means. Nondeterminism should be... introduced where needed...”
  • Closed nested transactions are (fairly) deterministic; open nested transactions introduce nondeterminism (and complexity!) in hopes of increased parallelism.
Contributions

• Contributions of the paper that we will (try to) cover:
  • new language constructs for open nesting (including abstract locks) and their semantics
  • discussion of open nesting pitfalls
  • performance evaluation of open nested transactional system for realistic data structures
Outline

- Preliminaries
  - McRT-STM: closed nesting w/ partial abort
  - Open nesting language extensions
Nested Transaction

- *NOTE*: this presentation (like the corresponding paper) restricts itself to *linear nesting*: a transaction can have only one live child transaction at a time.

- At the language level, lexically scoped transactions amount to linear nesting
McRT-STM

• Supports closed nested transactions with partial abort
• Strict two-phase locking(?) to detect write conflict
• Optimistic concurrency control w/ versioning for reads
McRT-STM

- Each thread owns a *transaction activation stack of transaction descriptors*

<table>
<thead>
<tr>
<th>$tk$</th>
<th>ReadSet</th>
<th>$(..., (ti, ni), ...)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WriteSet</td>
<td>$(..., (ti, ni), ...)$</td>
</tr>
<tr>
<td></td>
<td>UndoLog</td>
<td>$(..., (li, vi), ...)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$t_j$</th>
<th>ReadSet</th>
<th>$(..., (ti, ni), ...)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WriteSet</td>
<td>$(..., (ti, ni), ...)$</td>
</tr>
<tr>
<td></td>
<td>UndoLog</td>
<td>$(..., (li, vi), ...)$</td>
</tr>
</tbody>
</table>
- Added to each memory location is a *transaction record* consisting of a version number.

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$o_i.f_j$</td>
<td>$v$</td>
<td>$n$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
McRT-STM

• Locks are (conceptually) maintained in a lock table

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$o_i.f_j$</td>
<td>$t_k$ or none</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Running Example

```java
1: class IntSet {
2:   private int[] data = null;
3:   private int size;
4:   public IntSet(int maxSize) {
5:     data = new int[maxSize]; }
6:   public atomic void insert(int val) {
7:     if (size==data.length) return;
8:     for (int i=0; i<size; i++)
9:       if (data[i] == val) return;
10:    data[size] = val;
11:    size = size+1;
12:  }
13: }
14: public atomic boolean contains(int val) {
15:   for (int i=0; i<size; i++) {
16:     if (v==val) return true;
17:   }
18:   return false;
19: }
20:}

// thread 1
atomic {
  s.insert(10);
  s.insert(20);
}

// thread 2
atomic {
  s.insert(30);
}

IntSet s = new IntSet(MAX);
```
On transaction start, a new transaction descriptor with empty read/write sets and undo log is added to the transaction activation stack.

### Thread 1
1. txn start // $t_1$
2. txn start // $t_2$

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>none</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

### Thread 2
3. txn start // $t_3$

<table>
<thead>
<tr>
<th>Location</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadSet</td>
<td>()</td>
</tr>
<tr>
<td>WriteSet</td>
<td>()</td>
</tr>
<tr>
<td>UndoLog</td>
<td>()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadSet</td>
<td>()</td>
</tr>
<tr>
<td>WriteSet</td>
<td>()</td>
</tr>
<tr>
<td>UndoLog</td>
<td>()</td>
</tr>
</tbody>
</table>
When reading, if the location isn’t locked by another thread, add version to the read set.

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>none</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

Thread 1
1. txn start // t₁
2. txn start // t₂
4. read size

Thread 2
3. txn start // t₃
5. txn start // t₄
6. read size
When writing, if the location isn’t locked, lock it. Then update the location’s version and the write set.

**Thread 1**
1. txn start // \( t_1 \)
2. txn start // \( t_2 \)
4. read size
7. write data[0]=10
8. write size

**Thread 2**
3. txn start // \( t_3 \)
5. txn start // \( t_4 \)
6. read size
When reading or writing, if the location is locked, appeal to the contention manager.

### Thread 1

1. txn start // \( t_1 \)
2.txn start // \( t_2 \)
4. read size
7. write data[0]=10
8. write size=1

### Location

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

### Location

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>none</td>
</tr>
</tbody>
</table>

### Thread 2

3. txn start // \( t_3 \)
5. txn start // \( t_4 \)
6. read size
9. write data[0] // locked!
10. txn abort // \( t_4 \)

### Location

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

### Location

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>none</td>
</tr>
</tbody>
</table>
When aborting nested transaction:
1. Process undo log
2. Reset written version numbers
3. Pop transaction descriptor

Thread 1
1. txn start // $t_1$
2. txn start // $t_2$
4. read size
7. write data[0]=10
8. write size=1

Thread 2
3. txn start // $t_3$
5. txn start // $t_4$
6. read size
9. write data[0] // locked!
10. txn abort // $t_4$

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>$t_2$</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>$t_2$</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>
When committing nested transaction:
1. Validate read set
2. Merge logs/locks with parent
3. Pop transaction descriptor

### Thread 1
1. txn start // $t_1$
2. txn start // $t_2$
4. read size
7. write data[0]=10
8. write size=1
11. txn commit // $t_2$

#### Location

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>$t_1$</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>$t_1$</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>none</td>
</tr>
</tbody>
</table>

#### Value

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

### Thread 2
3. txn start // $t_3$
5. txn start // $t_4$
6. read size
9. write data[0] // locked!
10. txn abort // $t_4$

#### Location

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>0</td>
</tr>
<tr>
<td>WS</td>
<td>0</td>
</tr>
<tr>
<td>UL</td>
<td>0</td>
</tr>
</tbody>
</table>
(let’s let Thread 1 continue for a bit...) Notice there’s no need to add to read set a parent already has a write lock

Thread 1

1. txn start // $t_1$
2. ...
12. txn start // $t_5$
13. read size
14. read data[0]
15. write data[1] = 20
16. write size = 2

Thread 2

3. txn start // $t_3$
5. txn start // $t_4$
6. read size
9. write data[0] // locked!
10. txn abort // $t_4$

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>$t_1$</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>$t_1$</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>$t_5$</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>2</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>20</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>WS</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>UL</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Thread 1

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>2</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>20</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

Thread 2
Committing another closed nested transaction...

### Thread 1

1. txn start // $t_1$
2. ...
12. txn start // $t_5$
13. read size
14. read data[0]
15. write data[1]=20
16. write size=2
17. txn commit // $t_5$

### Thread 2

3. txn start // $t_3$
4. txn start // $t_4$
5. read size
9. write data[0] // locked!
10. txn abort // $t_4$

#### Locking Information

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>$t_1$</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>$t_1$</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>$t_1$</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>none</td>
</tr>
</tbody>
</table>

#### Locking Information

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>2</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>20</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

#### Locking Information

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>0</td>
</tr>
<tr>
<td>WS</td>
<td>0</td>
</tr>
<tr>
<td>UL</td>
<td>0</td>
</tr>
</tbody>
</table>
Thread 1

1. txn start // \(t_1\)

... 

12. txn start // \(t_5\)

13. read size

14. read data[0]

15. write data[1]=20

16. write size=2

17. txn commit // \(t_5\)

18. txn commit // \(t_1\)

Thread 2

3. txn start // \(t_3\)

5. txn start // \(t_4\)

6. read size

9. write data[0] // locked!

10. txn abort // \(t_4\)

When committing at top-level, additionally release all locks

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>none</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>2</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>20</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

\(t_1\)

\( RS \) ((size,v0))

\( WS \) ((data[0],v0),(size,v0),(data[1],v0))

\( UL \) ((data[0],0),(size,0),(data[1],0),(size,1))

\( t_3\)

<table>
<thead>
<tr>
<th>RS</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS</td>
<td>0</td>
</tr>
<tr>
<td>UL</td>
<td>0</td>
</tr>
</tbody>
</table>
Finally, let’s let Thread 2 complete its transaction...

Thread 1

1. txn start // t₁
   ...
12. txn start // t₅
13. read size
14. read data[0]
15. write data[1]=20
16. write size=2
17. txn commit // t₅
18. txn commit // t₁

Thread 2

3. txn start // t₃
   ...
19. txn start // t₆
20. read size
21. read data[0]
22. read data[1]
23. write data[2]=30
24. write size=3
Finally, let’s let Thread 2 complete its transaction...

1. txn start // $t_1$
2. ...
12. txn start // $t_5$
13. read size
14. read data[0]
15. write data[1]=20
16. write size=2
17. txn commit // $t_5$
18. txn commit // $t_1$

Thread 2
3. txn start // $t_3$
4. ...
25. txn commit // $t_6$

<table>
<thead>
<tr>
<th>Location</th>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>$t_3$</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>none</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>$t_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>3</td>
<td>v2</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>20</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>30</td>
<td>v1</td>
</tr>
</tbody>
</table>

RS $t_6$: $(((\text{size},v1), (\text{data}[0],v1), (\text{data}[1],v1)))$
WS $t_6$: $(((\text{data}[2],v0), (\text{size},v1)))$
UL $t_6$: $(((\text{data}[2],0), (\text{size},2)))$

RS $t_3$: $(((\text{size},v1), (\text{data}[0],v1), (\text{data}[1],v1)))$
WS $t_3$: $(((\text{data}[2],v0), (\text{size},v1)))$
UL $t_3$: $(((\text{data}[2],0), (\text{size},2)))$
Finally, let’s let Thread 2 complete its transaction...

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>3</td>
<td>v2</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>20</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>30</td>
<td>v1</td>
</tr>
</tbody>
</table>
Outline

- Preliminaries
- McRT-STM: closed nesting w/ partial abort
  - Open nesting language extensions
Open Nesting

• Open nesting can (sort of) be viewed as a generalization of closed nesting
  • closed nested transactions must obtain and manage write locks
  • before commit, closed nested transactions must validate optimistic reads
  • on abort, closed nested transactions must undo writes

• Open nesting generalizes locks, validation, and undo actions, giving the programmer more control
Open Nesting

- Read set is extended to contain programmer-defined validation handlers in addition to read entries
  - validation handlers are run as open nested transactions when attempting to commit
- Undo log is extended to contain programmer-defined abort handlers in addition to undo entries
  - abort handlers are run as open nested transactions when aborting
- Lock table is extended to support abstract locks
Language Extensions

- Open nesting is supported at the class level
  - a class can be designated openatomic
  - openatomic classes may contain only private fields
  - all public methods are implicitly openatomic as well
  - openatomic methods may contain any combination of the four handler types, and also optionally a locks clause defining abstract locks
Language Extensions

- **on-abort**: undo the effects of an action
- **on-validation**: check for access conflicts
- **on-commit**: (for example) write buffered effects
- **on-top-commit**: similar to on-commit
Language Extensions

- New expression forms for use in handlers
  - @result: the value originally returned by the method
  - @old(exp): the value of exp at the time the method was called
Abstract Locks

- **the locks clause** specifies abstract locks that must be obtained before the method can **return**
  - authors are purposefully vague about when locks are **actually** acquired; we’ll assume the attempt is made at commit time

- **an abstract lock consists of:**
  - a *context object*: the lock holder
  - a *locked object*: can be any object; stands for the abstract concept that is being ‘locked’
  - a *lock mode*: comes from a particular lock mode class
Abstract Locks

• Lock modes and compatibility matrix used in the paper:

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>IX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>IX</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>X</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

• When consulting the lock table to see if an abstract lock is already held:
  • the context objects are compared for identity
  • the locked objects are compared for equality
Extending McRT-STM

• Read sets and undo logs may now contain arbitrary code blocks (the handlers)
• The lock table may contain abstract locks as well as write locks
• New lists of on-commit/on-top-commit handlers are maintained (but our example doesn’t use them)
• On commit, an open nested transaction passes on its handlers, but not its normal read/write set entries, undo entries, or write locks
IntSet Revisited

1: openatomic class IntSet {
2:   private int[] data = null;
3:   private int size;
4:   private final Object lock = new Object();
5:   public IntSet(int maxSize) {
6:     data = new int[maxSize]; }

7:   public boolean insert(int val) {
8:     locks(new Integer(val):X, lock:IX) {
9:       if (size==data.length) return false;
10:      if (indexOf(val) == -1) return false;
11:     data[size] = val;
12:     size = size+1;
13:     return true;
14:   } onabort {
15:     if (@result) { /* undo code */ }
16:   }

17:   public boolean contains(int val) {
18:     locks(new Integer(val):S) {
19:       return !(indexOf(val)==-1);
20:     }
21:   }
22: }

... IntSet s = new IntSet(MAX);

• indexOf() is private, returns -1 if value is not found
• the /* undo code */ needs to shift left all array elements after the index of the value being removed, and update size

// thread 1
atomic {
  s.insert(10);
  s.insert(20);
}

// thread 2
atomic {
  s.insert(30);
}
Thread 1 begins as before.

Thread 2
1. txn start // \(t_1\)
2. txn start // \(t_2\)
3. read size
4. write data[0]=10
5. write size=1

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>(t_2)</td>
<td></td>
</tr>
<tr>
<td>s.data[0]</td>
<td>(t_2)</td>
<td></td>
</tr>
<tr>
<td>s.data[1]</td>
<td>(v_0)</td>
<td></td>
</tr>
<tr>
<td>s.data[2]</td>
<td>(v_0)</td>
<td></td>
</tr>
</tbody>
</table>
On commit of open nested transaction:
1. read set validated
2. undo log cleared
3. write locks cleared
4. abstract locks obtained
5. handlers appended to parent

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>1</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>0</td>
<td>v0</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

**Thread 1**
1. txn start // t1
2. txn start // t2
3. read size
4. write data[0]=10
5. write size=1
6. txn commit // t2

**Thread 2**

```
1. txn start // t1
2. txn start // t2
3. read size
4. write data[0]=10
5. write size=1
6. txn commit // t2
```
At this point, Thread 2 may perform an insert provided the inserted value is not 10.

---

### Thread 1

1. txn start // $t_1$
2. txn start // $t_2$
3. read size
4. write data[0]=10
5. write size=1
6. txn commit // $t_2$

---

### Thread 2

7. txn start // $t_3$
8. txn start // $t_4$
9. read size
10. read data[0]
11. write data[1]=30
12. write size=1

---

### Location Table

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>1</td>
<td>v2</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>30</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

---

### Thread 1

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>()</td>
</tr>
<tr>
<td>WS</td>
<td>()</td>
</tr>
<tr>
<td>UL</td>
<td>(delete(10))</td>
</tr>
</tbody>
</table>

---

### Thread 2

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>()</td>
</tr>
<tr>
<td>WS</td>
<td>()</td>
</tr>
<tr>
<td>UL</td>
<td>()</td>
</tr>
</tbody>
</table>
Thread 2 may commit its open nested transaction, immediately publishing the results.

### Thread 1
1. txn start // \(t_1\)
2. txn start // \(t_2\)
3. read size
4. write data[0]=10
5. write size=1
6. txn commit // \(t_2\)

### Thread 2
7. txn start // \(t_3\)
8. txn start // \(t_4\)
9. read size
10. read data[0]
11. write data[1]=30
12. write size=1
13. txn commit // \(t_4\)

---

#### Table: Location

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>2</td>
<td>v2</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>30</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>0</td>
<td>v0</td>
</tr>
</tbody>
</table>

#### Table: RS

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>(size,v1),(data[0],v1)</td>
</tr>
</tbody>
</table>

#### Table: WS

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS</td>
<td>((data[1],v0),(size,v1))</td>
</tr>
</tbody>
</table>

#### Table: UL

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UL</td>
<td>((data[1],0),(size,1))</td>
</tr>
</tbody>
</table>

---

#### Table: CTX

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>10</td>
<td>X</td>
</tr>
<tr>
<td>s</td>
<td>lock</td>
<td>IX</td>
</tr>
<tr>
<td>s</td>
<td>30</td>
<td>X</td>
</tr>
<tr>
<td>s</td>
<td>lock</td>
<td>IX</td>
</tr>
</tbody>
</table>

---

#### Thread 1

- RS: ()
- WS: ()
- UL: (delete(10))

#### Thread 2

- RS: ()
- WS: ()
- UL: (delete(30))
Thread 2 may also commit its top-level transaction, releasing the abstract locks

Thread 1
1. txn start // $t_1$
2. txn start // $t_2$
3. read size
4. write data[0]=10
5. write size=1
6. txn commit // $t_2$

Thread 2
7. txn start // $t_3$
8. txn start // $t_4$
9. read size
10. read data[0]
11. write data[1]=30
12. write size=1
13. txn commit // $t_4$
14. txn commit // $t_3$
Thread 2’s insert does not abort Thread 1’s closed nested transaction \( t_1 \), which may complete without incident.
Thread 2’s insert does not abort Thread 1’s closed nested transaction \( t_1 \), which may complete without incident.
Thread 2’s insert does not abort Thread 1’s closed nested transaction \( t_1 \), which may complete without incident.
Thread 1

.Done!

Thread 2

7. txn start // $t_3$
8. txn start // $t_4$
9. read size
10. read data[0]
11. write data[1]=30
12. write size=1
13. txn commit // $t_4$
14. txn commit // $t_3$

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.size</td>
<td>3</td>
<td>v3</td>
</tr>
<tr>
<td>s.data[0]</td>
<td>10</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[1]</td>
<td>30</td>
<td>v1</td>
</tr>
<tr>
<td>s.data[2]</td>
<td>20</td>
<td>v1</td>
</tr>
</tbody>
</table>
Outline

- Preliminaries
- McRT-STM: closed nesting w/ partial abort
- Open nesting language extensions
Evaluation/Discussion

• Our examples showed that open nesting can be used to allow logically correct interleavings that closed nesting forbids

• But did open nesting really increase parallelism?
  • Notice that the open nested inserts performed by the two threads would still conflict with one another if they executed concurrently

• We skipped over a lot! Any questions?