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

```c
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 \( s \)?

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:

Abstract view: \{10,20,30\} \quad \{10,30\}

Physical view:

physically 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>ti</th>
<th>ReadSet</th>
<th>(...(t,v),...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WriteSet</td>
<td>(...(t,v),...)</td>
</tr>
<tr>
<td></td>
<td>UndoLog</td>
<td>(...(l,v),...)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tj</th>
<th>ReadSet</th>
<th>(...(t,v),...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WriteSet</td>
<td>(...(t,v),...)</td>
</tr>
<tr>
<td></td>
<td>UndoLog</td>
<td>(...(l,v),...)</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>a.fj</td>
<td>ν</td>
<td>n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

• 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>a.fj</td>
<td>t₀ 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:   }
7:   public atomic void insert(int val) {
8:     if (size==data.length) return;
9:     for (int i=0; i<size; i++)
10:       if (data[i] == val) return;
11:     data[size] = val;
12:     size = size+1;
13:   }
14:   public atomic boolean contains(int val) {
15:     for (int i=0; i<size; i++)
16:       if (v==val) return true;
17:     return false;
18:   }
19: }
20:}
```

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

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

On transaction start, a new transaction descriptor with empty read/write sets and undo log is added to the transaction activation stack

<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>

Thread 1
1. txn start // t₁
2. txn start // t₂

Thread 2
3. txn start // t₃
When reading, if the location isn't locked by another thread, add version to the read set.

When writing, if the location isn't locked, lock it. Then update the location's version and the write set.

When reading or writing, if the location is locked, appeal to the contention manager.

When aborting nested transaction:
1. Process undo log
2. Reset written version numbers
3. Pop transaction descriptor
When committing nested transaction:
1. Validate read set
2. Merge logs/locks with parent
3. Pop transaction descriptor

### Thread 1
- 1. txn start // t
- 2. txn start // t2
- 3. txn start // t3
- 4. read size
- 5. read size
- 6. write data[0]=10
- 7. write size=1
- 8. write data[0]=10
- 9. write data[0] // locked!
- 10. txn abort // t4

### Thread 2
- 1. txn start // t
- 2. txn start // t2
- 3. txn start // t3
- 4. read size
- 5. write data[0]
- 6. write size=1
- 7. write data[0] // locked!

(lets let Thread 1 continue for a bit...)

Notice there's no need to add to read set a parent already has a write lock.

When committing at top-level, additionally release all locks.

### Committing another closed nested transaction...

### Thread 1
- 1. txn start // t
- 2. txn start // t2
- 3. txn start // t3
- 4. read size
- 5. read size
- 6. write data[0]=20
- 7. write size=2
- 8. write data[0] // locked!
- 9. txn abort // t4

### Thread 2
- 1. txn start // t
- 2. txn start // t2
- 3. txn start // t3
- 4. read size
- 5. write data[1]
- 6. write size=2
- 7. write data[1] // locked!
- 8. txn abort // t4

### When committing at top-level, additionally release all locks

### Thread 1
- 1. txn start // t
- 2. txn start // t2
- 3. txn start // t3
- 4. read size
- 5. read size
- 6. write data[0]=20
- 7. write size=2
- 8. write data[1] // locked!
- 9. txn abort // t4

### Thread 2
- 1. txn start // t
- 2. txn start // t2
- 3. txn start // t3
- 4. read size
- 5. write data[0]
- 6. write size=2
- 7. write data[0] // locked!
- 8. txn abort // t4
Finally, let's let Thread 2 complete its transaction...

### Thread 1
1. txn start // t1
   ...
12. txn start // t6
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
   ...
12. txn start // t6
13. read size
14. read data[0]
15. write data[1]=20
16. write size=2
17. txn commit // t5
18. txn commit // t1

### Location Value

<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>

### Lock Holder

<table>
<thead>
<tr>
<th>Lock Holder</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts</td>
</tr>
<tr>
<td>t6</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

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

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

- 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

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

Abstract Locks

- Lock modes and compatibility matrix used in the paper:

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>x</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>x</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
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:   }
8:   public boolean insert(int val) {
9:     locks(new Integer(val):X, lock:IX) {
10:    if (size==data.length) return false;
11:    if (indexOf(val) == -1) 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:  ...
23:  }
24:  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 begins as before.

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

Thread 2

At this point, Thread 2 may perform an insert provided the inserted value is not 10

On thread commit:
1. txn start // t1
2. txn start // t2
3. read size
4. write data[0]=10
5. write size=1
Thread 2 may commit its open nested transaction, immediately publishing the results:

Thread 2 may also commit its top-level transaction, releasing the abstract locks.

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 $t_2$, which may complete without incident.

**Thread 1**

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**

... 22. txn commit // $t_1$

---

**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?