(Nested) Transactions:

Closed, Open, and Boosted

J. Eliot B. Moss

Professor, University of Massachusetts

moss@cs.umass.edu

Transactions are Good

- They deal with concurrency
  - Atomic txns avoid problems with locks
    - Deadlock, wrong lock, priority inversion, etc.
- They handle recovery
  - Retry in case of conflict
  - Cleanup in face of exceptions/errors

More practical for ordinary programmers than locks to code robust concurrent systems

Semantics of Transactions

- They offer A,C,I of database ACID properties:
  - Atomicity: all or nothing
  - Consistency: single global order
  - Isolation: intermediate states invisible
- In sum, ***serializability***, in face of concurrent execution and transaction failures
- Can be provided by *Transactional Memory*
  - Hardware, software, or hybrid

Simple Transactions for Java

Following Harris and Fraser, we might offer:

atomic { S }

- Atomic: Execute **S** entirely or not at all
- Isolated: No other atomic action can see state in the middle, only before **S** or after it
- Consistent: All other atomic actions happen logically before **S** or after **S**

Implement with r/w locking/logging, on words or whole objects; optimistic, pessimistic, etc.

A Basic Software Implementation

- Add a version number / owner to each object
- Each read records (read, object, version) in a log associated with the transaction
- Each write causes this transaction to try to become the object's owner (use compare-and-swap or similar); records (write,object, version) in the log
- Each write records (object, field, old-value) in the transaction's log

Basic Implementation (continued)

- May commit if each object *read* has same version or is owned by this transaction
- Commit increments version number of each object owned by this transaction

Failure / abort:

- Apply write log entries in reverse order
- Release ownership of objects, restoring original version number

Basic Implementation (3)

Properties of this basic strategy:

- Reads are *optimistic* – record version number and *validate* at the end; avoids writes/locks on the object itself
- Writes are *pessimistic* – grab "lock" eagerly
- *Update-in-place* writing strategy
  - … implying *undo log* failure strategy

Why is this better than locking?

- ***Abstract:*** Expresses intent without over- or under-specifying how to achieve it: *correct*
- ***Allows unwind and retry:*** More flexible response to conflict: *prevents deadlock*
- ***Allows priority without deadlock:*** Avoids *priority inversion* (still need to avoid livelock)
- ***Allows more concurrency:*** synchronizes on exact data accessed rather than an object lock*

  … and distinguishes read and writes

* The basic strategy is intermediate in granularity

Limitations of simple transactions

- Long/large transactions either reduce concurrency or are unlikely to commit
- Data structures often have false conflicts
  - Reorganizing tree nodes

Closed Nesting

Model proposed in 1981 (Moss PhD):

- Each subtxn builds its own read/write set
- On commit, merge with its parent's sets
- On abort, discard its set
- Subtxn never conflicts with ancestors
  - Conflicts with *non-ancestors*
  - Can see ancestors' intermediate state, etc.
- Requires keeping values at each nesting level that writes a data item

Closed Nesting Example

```
atomic {
        A1;
        atomic { s11; s12; }
        A2;
        atomic { s21; s22; s23; }
        A3;
}
```

Closed Nesting Helps: Partial Rollback

- When actions conflict, one will be rolled back
- With closed nesting, roll back only up through the youngest conflicting ancestor
- This reduces the amount of work that must be redone when retrying

Limitations of Closed Nesting

Limitations of closed nesting derive from the original non-nested semantics:

- *Aggregates larger and larger conflict sets*
  - Still hard to complete long/large txns
- *Synchronizes at physical level*
  - Gives false conflicts

Open nesting to the rescue!

A concept and theory developed in the 1980s

- Comes from the database community
- Partly an explanation/justification of certain real strategies employed in database systems
- Partly an approach to generalizing those strategies

Conceptual Backdrop of Open Nesting

- Closed nesting has just one level of abstraction:
  - *Memory contents*
    - Basis for concurrency control
    - Basis for rollback
- Open nesting has *more levels of abstraction*
- Each level may have a distinct:
  - *Concurrency control model* (style of locks)
  - *Recovery model* (operations for undoing)

Open Nested Actions

- While running, a leaf open nested action
  - Operates at the memory word level
- When it commits:
  - Its memory changes are *permanent*
  - Concurrency control and recovery *switch levels*
    - Give up memory level "locks":

      acquire *abstract locks*
    - Give up memory level unwind

      use only *inverse operations* (undos)

Non-Leaf Open Nested Actions

- A non-leaf open nested action
  - Operates at the memory word level, *and*
  - May accumulate abstract locks and undos from committed children
- When it commits:
  - Its memory changes are permanent
  - Concurrency control and recovery switch levels
    - Give up memory level "locks" *and* child locks:
      acquire *abstract locks* for new level
    - Give up memory level unwind *and* child undos
      use only inverses (undos) for new level

Open Nesting and Data Abstraction

Open nesting naturally fits *types*, not code chunks

- For safety, memory state accessed by an open action must not be accessed by closed actions
- Abstract data types neatly encapsulate state
- Data types also tend to provide inverses
- Abstract locks match abstract state/operations

Simple application: Phone directory

- Employee phone directory
  - Name-to-number lookup
  - All names in a range
  - All entries in a department

- Structure: Uses balanced trees
  - Tree to map names to records
  - Tree to map depts to sets of records

Layers of abstraction

- *Phone directory:* top (most abstract) layer
  - Insert must create record, add to two trees
  - Delete must remove from two trees
  - Desire high concurrency
- *(Indexed) set of records:* middle layer
  - Central notion: presence/absence of records in sets
- *Tree:* lowest layer:
  - Tree nodes and pointers to records

A scenario: Concurrent insertions

- Two transactions, inserting different names
- Close in alphabet, so same tree node
- Conflict at level of read/write sets (words)
- "Early commit" of the two tree inserts is ok
  - Each insert is atomic: if not, breaks tree!
  - Different names, so no *abstract conflict*

    That is, at the level of a set of (key, value) pairs
- ***But*** … entails some *obligations*

Open actions need *abstract* undo

Start: “Sloan” in node

Open action 1 adds “Smith”, commits

Open action 2 adds “Smythe”, commits

Parent of 1 aborts, smashes node!

Tree node

Same example with *abstract* undo

Start: “Sloan” in node

Open action 1 adds “Smith”, commits

Open action 2 adds “Smythe”, commits

Parent of 1 aborts, *deletes* “Smith”

Tree node

What is a correct undo?

- Consider abstract state
  - Here: set of (name,phone) pairs
  - Ordered by name in tree
  - Etc.
- Insert: goes from “without name” to “with”
- Undo must restore pre-insert (abstract) state when presented with the post-insert (abstract) state

What is a *good* correct undo?

- One that minimizes concurrency conflicts
- So, in this case, concerned only with presence/absence of the inserted name
- Thus: delete(…) is a good undo here

But wait! There's more ….. !

A different scenario

Start: “Sloan” in node

Open action 1 adds “Smith”, commits

Open action 2 sees “Smith”, commits

Parent of 1 aborts, removes “Smith”

1 2	Sloan	Smith	

Tree node

The concurrency control obligation

Problem: Allowed uncommitted data to be seen:

too much concurrency!

Why is this a problem?

Open action 2 saw a “phantom” value

This is *not serializable!*

How to regain (abstract) serializability

- Tx holds an *abstract lock* to indicate that the entry is in doubt until Tx commits
  - Ty (child) says what this lock should be; the level shifts as Ty commits
- Might add a "pending" flag to records
  - Check it when accessing/deleting a record
- Similar technique needed for deletes

This *almost* works, but ….

Another concurrency scenario

Start: “Sloan” in node

Open action 1 sees “Smith” is absent

Top action (2) adds “Smith”, commits

Open action 1 sees “Smith” is *present*

1 2	Sloan	Smith	

Tree node

Concurrency control is subtle!

No transaction isolation!

Action 1 should have “locked” *absence* of “Smith”

In general, need an abstract lock data structure

We can remember locked *keys* in a side table

S (share) and X (exclusive) modes

Failing lookup locks “Smith”, so insert will conflict

Another concurrency scenario

Start: “Sloan” in node

Open action 1 sees “Smith” is absent

Open action 2 desires to add “Smith”

Tries to lock “Smith” X mode — *fails*

Tree node

Abstract locks

Putting it together

To insert “Smith”:

1. Acquire X mode lock on key “Smith”
2. Insert in by-name tree
3. Insert in department tree

- To commit:
  - Release abstract lock
- To abort:
  - Delete from dept tree, then by-name
  - Release abstract lock

Looking up a name

To look up “Smith”:

1. Acquire S mode lock on key “Smith”
2. Look up in by-name tree

   Returns null if absent, record if present

- To commit:
  - Release abstract lock
- To abort:
  - Release abstract lock

End result

- Insertions, etc., can be “pipelined”
  - Good concurrency, yet tree is safe
  - Can also pipeline through layers of a tree
  - Inherent, i.e., abstract, conflicts respected
- Concurrency control now at abstract level
- Undos also at abstract level

Primer on abstract state

- Some (not all) concrete states *s* are *valid*
  - Example: tree ordered, no duplicates
- Every valid concrete *s* maps to an abstract *S*
  - Example: tree maps to {(key,value)}
- Abstraction map defines *equivalence classes*
  - Concrete states *s* that map to same *S*
- Helpful to design in terms of abstraction map, if only informally, and to document it

Abstract Serializability

- Lock parts of abstract state
- Undo in the abstract

Result is *abstract serializability*

- Undo restores changed part of abstract state
- Lock must prevent conflicting forward ops
- Lock must insure undo remains applicable

Pieces fit with each other

Data type works correctly *as a whole:*

- Protected concrete state
- Clearly understood abstract state
- Abstract locks, in terms of abstract state
- Abstract undos, in terms of abstract ops

How to *implement* open nesting?

- Parent maintains *abort*, *commit*, and *done* action lists
- Commit of an open nested action adds:
  - Undo to the *abort* list
  - Unlock to the *done list*
  - Cleanups (if any) to the *commit* list

  Sometimes better to change state lazily; e.g., delete late to hold space until sure

Commit and abort semantics

- When parent *commits:*
  - Run *commit* actions, then
  - Run *done* actions (and do r/w sets)

- When parent *aborts:*
  - Run *abort* actions, then
  - Run *done* actions (and do r/w sets)

"The log is the truth"

Aborting is a little more subtle …

- An undo should be applied in the state that held when its forward action committed
- Consider:

  memory A, open B, memory C, open D
- State for D^{-1} should see A and C
- State for B^{-1} should see A but *not* C
- Abort = D^{-1}, undo C, B^{-1}, undo A

  Can do this using levels of closed nesting

Boosting

- Open nesting is built on top of an assumed software transactional memory (or hardware, or hybrid)
- Boosting is built on top of assumed "thread-safe" (technically: *linearizable*) data types
  - Implement insides of a concurrent data structure however you like, as long as it's concurrency safe
  - Make it *transactional* by wrapping it with abstract locks and abstract undos

Properties of Boosting

- Leverages existing implementations
- These are often “best of breed”
- May perform better than open nesting
- May require stronger abstract locks
  - Must acquire *before* performing operation
  - May need to over-acquire, e.g., X mode if *trying* to add, while open nesting can acquire in S mode if already present
    - Boosting could adjust lock after the fact

Transactional Foo vs Concurrent Foo

- *Concurrent* data type allows *one operation at a time* from each thread to be atomic
- *Transactional* data type allows a *composed group of operations* to be atomic

One Approach to Abstract Locks

- Exploit the analogy/metaphor of geometric *shapes* within a shared *space*
  - Two locks *overlap* if they lie in the same space and their shapes overlap
- To this, add the traditional notion of lock *modes*:
  - Two locks *conflict* if they overlap and their modes conflict

Shape Examples

- Unordered set items:
  collection of discrete *points*
  - Can organize these in a hash table

- Ordered set items:
  collection of discrete points +
  open and closed *ranges* (for range tests) +
  open and close *rays* (for ranges at ends)
  - Can organize these in a tree structure

Lock Mode Examples

- X (exclusive) by itself (Java synchronized)
- S (shared) + X – read/write locks
- Pin + Change:
  - Pin ok with Pin; Change ok with Change
  - Pin conflicts with Change
  - For a counter:
    - Incr/Decr → Change mode
    - Read → Pin mode
  - *Not the same as S + X modes*

A Challenge in Abstract Locking

- Even “read” (S mode) locks lead to conflicts on inserting and removing in the *abstract locking* data structure itself
- A possible solution is to use something like open nesting on this data structure
  - But this does not work if you are using today’s Hardware TM to update the data structure
  - Certain cases can be optimized, but not the general case

Can ordinary programmers use this?

- Single-level and closed nesting usually enough
  - Most programmers don't need open nesting
- Open nesting good for library classes
  - High concurrency, or special semantics
- Our experience is:
  - Undos are usually trivial to provide
  - Other clauses not often necessary
    - Assuming lock release is implied
  - Abstract locking takes getting used to
    - Fertile ground for library work

What are *we* up to?

- A reference implementation of nesting, open and closed, also with boosting and HTM (!) for Java

- Given abstract model of data type and operations:
  - Verify concurrency control
  - Verify inverses
  - Do this using automated proofs (SAT solver)

Parting Shots

- Nesting is desirable, open nesting needed
- Open nesting may be for the experts
- Need to integrate:
  - Desired semantics
  - Language design (with exceptions, etc.)
  - Run-time support
  - Memory level semantics
  - Hardware implementation

Partial Intellectual Genealogy

- C T Davies, *Data Processing Spheres of Control*, IBM Systems Journal, 1978 (root perhaps back to 1967 or before!)
- Eliot Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, PhD dissertation, MIT, 1981
- Herlihy & Moss, *Transactional Memory: Architectural Support for Lock-free Data Structures*, ISCA, 1993
- Harris & Fraser, *Language Support for Lightweight Transactions*, OOPSLA, 2003
- Moss & Hosking, *Nested Transactional Memory: Model and Architecture Sketches*, Sci. Comput. Prog., 2006
- Yang Ni *et al*, *Open Nesting for STM*, PPoPP 2007
- Herlihy & Koskinen, *Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects*, PPoPP 2008
- Chapman *et* al, *Closed and Open Nested Atomic Actions for Java: Language Design and Prototype Implementation*, PPPJ '14

Thank you! Questions?

Architecture and Language Implementation Lab

College of Information and Computer Sciences

University of Massachusetts Amherst

http://www.cs.umass.edu/~moss

Thanks to Tony Hosking and several students:

UMass: Trek Palmer

Purdue: Athul Acharya, Keith Chapman, and Phil McGachey

Talk to me also about …

- Garbage collection
  - Limits of GC performance
  - Applying machine learning to schedule GC
- Micro VM (Mu) – working on formal spec
- The chaos of computer performance

Recap: Why nest?

- To allow nesting of program constructs
  - *Can* just merge inner into outer …
  - But may induce more retry work
- To support multiple rollback/retry points
- To implement alternate strategies
- To increase concurrency (open)
- To offer selective permanence (open)
- To provide a general "escape hatch" (open)

Closed Nesting Helps: CCRs

Partial rollback helps *Conditional Critical Regions:*

Harris and Fraser's construct:

atomic (P) { S }

- Evaluate **P**, and if true, do **S** – all atomically
- If **P** is false, retry
  - Can "busy wait"
  - Can be smarter: wait until something changes that **P** depends on
    - Detect via conflict (give self lowest priority)

Closed Nesting Helps: Alternatives

One can try *alternatives:*

- When an action fails in a non-retriable way
- After some number of retries

Sample syntax:

atomic { S1 } else { S2 }

atomic (retries<5) { S1 } else { S2 }

Closed Nesting Helps: Concurrency

Subtransactions provide *safe* concurrency within an enclosing transaction

- Subtxns apply suitable concurrency control
- Subtxns fail and retry independently
- Great for mostly non-conflicting subactions
  - Tiles of a large array
  - Irregular concurrent computations
  - Replication in distributed systems

Thinking at the memory level

- Open nested action builds up r/w sets just like a closed nested action
- If open nested action aborts, discard sets, just like closed nested action
- If open nested action *commits:*
  - Install its writes, immediately, into the "global committed value"
  - If any ancestor holds that word, update it (but leave undo as is)

Properties of this rule

- Immediacy of update:
  - Ancestors (and others) see new value
- No concurrency surprises
  - Ancestors retain r/w sets (with new value)
  - Ancestors undo to their old value
- Note: Parent does not normally share global data with open nested child (encapsulation)
  - Example: B-tree nodes visible only to B-tree operations

What might the programmer write?

Something like:

```
atomic { S }
    onabort { A }
    oncommit { C }
    ondone { D }
```

- Open semantics implied by **onabort**, etc.
- Glossing over details: not a complete design
  - Need to deal with binding of variables, etc.

Proposal: Class-Based Open Atomic

Something like:

```
open atomic class NameSet {
    private Set<String> names = …;
   boolean insert(String name) throws Present
      locks(names, name, Exclusive) {
      if (names.contains(name))
     throw new Present();
      names.add(name);
    } onabort { names.remove(name); }
```

- Group at Intel MRL has a running implementation
- Substantial performance boost on micro-benchmarks

Bending the Rules

- Can use “improper” abstract locking to offer controlled communication
  - Can probably simulate Java wait/notify, e.g.
- Can use “improper” undo to cause truly permanent effect
  - Logging attempt to use a stolen credit card
  - But rolling back the rest of the transaction
- A general loophole: handy, but admittedly a dangerous “power tool”: use sparingly!