Sequential Specification of Transactional Memory Semantics

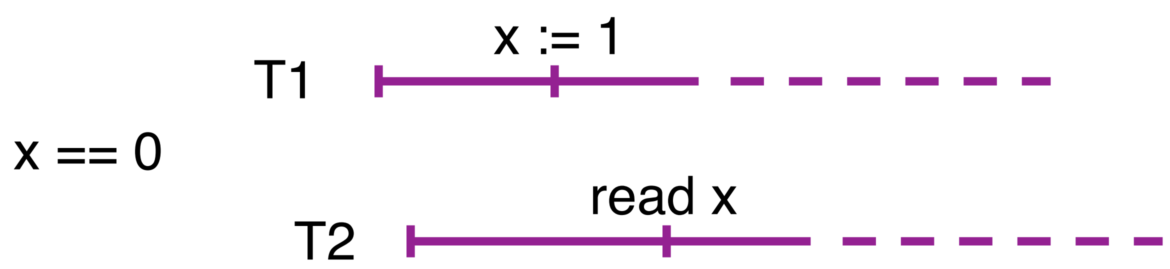
Michael L. Scott

University of Rochester

www.cs.rochester.edu/~scott/synchronization/

TRANSACT, Ottawa, June 2006

What Should TM Do?



- Read 0, in the hope that T2 will finish first?
- Read 1, on the assumption that T1 will finish first?
- Announce a conflict?
  - abort T1?
  - abort T2?
  - make T2 wait for T1's outcome?

Who cares?

- DB people understand these issues, in their world
- TM world is different
  - asynchronous concurrent access to shared memory
  - nonblocking concurrent objects; linearizability
- Formalism may help us
  - understand our systems better
  - prove them correct
  - identify new policies (stay tuned)

Nonblocking Concurrent Objects

- Stacks, queues, etc., but also locks, barriers, *and transactional memory*

- History = sequence of invocation and response events
- Sequential history = no overlap — each invocation immediately followed by corresponding response
- Sequential semantics = prefix-closed set of valid sequential histories

Example: Concurrent Queue

- Model state as unbounded sequence of enqueue and dequeue operations
- Sequential semantics = set of all histories in which
  - every enqueue returns true
  - if *n* previous dequeues have returned values, the next returns the value provided by the *n*th enqueue, if there have been that many, $\perp$ O.W.

✯ Don't know if concurrent implementation is correct unless we know how sequential implementation should behave

Transactional Memory

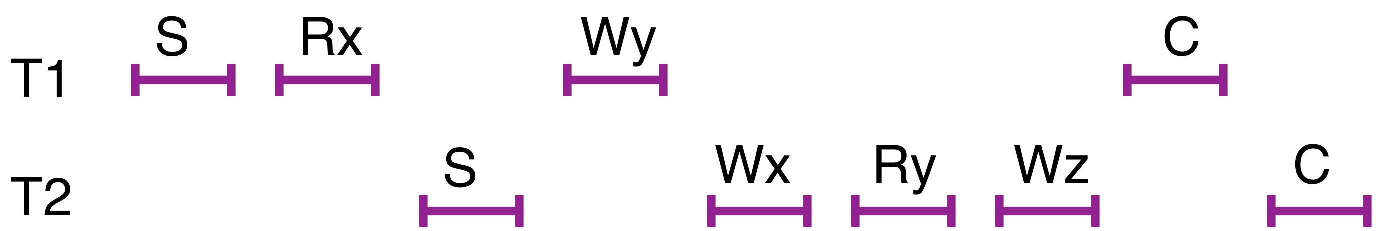
- Traditionally, a mechanism for easy implementation of correct concurrent objects
- BUT can also view as a concurrent object in its own right:
  - void start(t)
  - value read(o, t)
  - void write(o, d, t)
  - bool commit(t)
  - void abort(t)

  where t is the descriptor for (an instance of) a transaction

- In any correct implementation each of these methods will appear to be atomic — but that isn't any more complete a spec. than saying enqueue and dequeue are atomic

Well-formedness

- Require each thread subhistory to be a prefix of T*, where T = S (R|W)* (C|A)
- Operations do not overlap in a sequential history, but *transactions* do:



- History is *serial* if its transactions are *isolated* (do not overlap)
- Commit is *successful* if it returns true; transaction is successful if it ends with a successful commit

Consistency

- Require that
  - » each read returns the value written by the most recent successful transaction (or $\perp$ if none)
  - » values read are still *valid* at commit time
- This is more restrictive than absolutely necessary (doesn't permit speculative use of not-yet-committed writes), but seems reasonable for TM.

Fundamental TM Theorem

- If H is a consistent (well-formed) history, then so is the serial history J containing the successful transactions of H in commit order.
  - First drop all the unsuccessful transactions to get history I. Since consistency makes no mention of these, I remains consistent.
  - Now serialize I to get J. Since reads remain valid at commit time, J is consistent.
- Consistency of reads means TM *avoids cascading aborts;* fundamental theorem means TM is *strictly serializable.*

What are the valid sequential histories?

1. Must be consistent
2. Seems reasonable to require isolated transaction that ends with commit to succeed

- But when transactions overlap, which have an "excuse" to fail? Which, if any, must succeed?

Conflict functions

- $C(H, s, t)$ = true if s and t conflict in history H
- Require
  - » $C(H, s, t) = C(H, t, s)$
  - » s is isolated —> $\forall t\ C(H, s, t)$ = false
  - » $C(H, s, t) = C(I, s, t)$ if s and t interleave in the same way in H and I
- History is *C-respecting* if
  - » $C(H, s, t)$ = true —> at most one of s and t succeeds
  - » $[\forall t \sim C(H, s, t)]$ —> if s ends with commit, it succeeds
- *C-based TM* consists of all C-respecting histories
  - » can prove it is prefix-closed, and thus a sequential spec.

Simple conflict functions

Overlap conflict: C(H, s, t) = true if s and t overlap

Writer overlap conflict: C(H, s, t) = true if s and t overlap and at least one of them performs a write before the other ends

- These ignore *which objects* are read/written; seems appealing to refine that
- We will in general insist that conflict functions be *validity-ensuring*; that is, C(H, s, t) = true if s reads o, then t commits, having written o

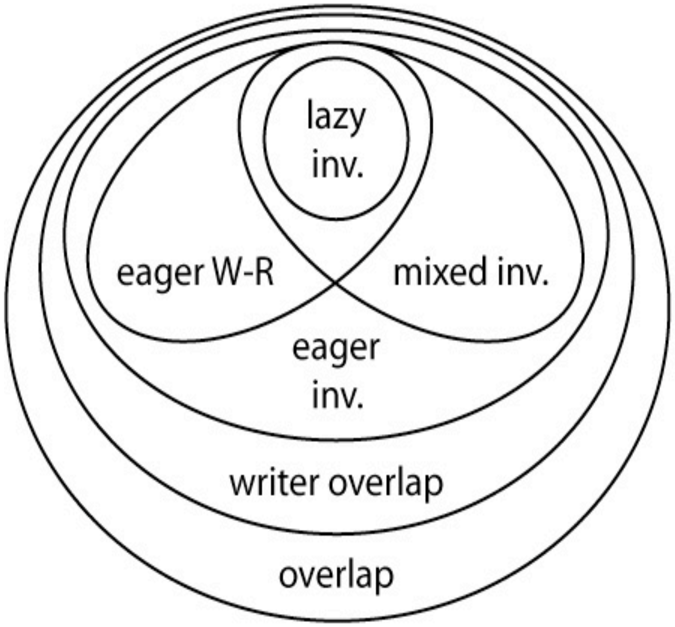
Object-based CFs

- Lazy: weakest possible; used in OSTM
- Eager: DSTM with visible readers
- Eager W-R: eager, but the write must happen first
- Hash-based variants: equivalence sets of objects

✯ Mixed invalidation: eager WW detection; lazy RW detection

CF implications

- Understand behavior of implementations
- Prove correctness
- Compare conflict detection strategies
  - » nontrivial relationships
  - » *Sets of histories* generally incomparable
  - » RSTM (described this morning) provides the 4 inner options (among others); currently exploring adaptation



But conflict isn't enough

- Conflict functions *give* transactions excuses to fail
  - » This admits implementations in which *all* conflicting transactions fail, or in which the "wrong one" or the "same one" always does
- Can show that any validity-ensuring CF admits livelock and starvation
- Can we address this by *requiring* some (non-isolated) transactions to succeed?

Arbitration functions

- A(H, s, t) = true if s and t conflict and s must fail
- Require
  - » A(H, s, s) = undefined
  - » A(H, s, t) = A(I, s, t) if H and I have the same prefix
- Typically A(H, s, t) = ~A(H, t, s), for s ≠ t (not required)
- History is *AC-respecting* if
  - » [C(H, s, t) = true ∧ A(H, s, t) = true] —> s fails
  - » [∀s ~C(H, s, t) ∨ A(H, s, t) = true] —> if t ends with commit, it succeeds
- *AC-based TM* consists of all AC-respecting histories

Candidate AFs

eagerly aggressive arbitration: whoever started most recently wins

lazily aggressive arbitration: whoever tries to commit first wins

Thm: eagerly aggressive, overlap-based TM is nonblocking

Thm: lazily aggressive C-based TM is livelock-free $\forall C$

Thm: lazily aggressive C-based TM admits starvation if C is validity ensuring

Real systems

- OSTM is lazily aggressive lazy invalidation-based, hence livelock-free but starvation-admitting
- DSTM (like other obstruction-free systems) is livelock-admitting

- Contention management (CM) in obstruction-free systems defers arbitration to the implementation
  - » simpler sequential semantics
  - » CM can consider priorities, load, etc.; can use randomization for probabilistic guarantees

Open questions

- Is it ever ok for a read to return the "wrong" value (if its transaction is doomed) or a speculative value?
- Can we characterize the CFs and AFs that admit/preclude livelock?
- How fancy can arbitration reasonably be?
  Can it be probabilistic? Can it preclude starvation?
- Are there reasons *not* to defer to CM?
- Would it be useful to allow ABA writes to weaken the notion of a validity-ensuring CF?
- Should non-overlapping txns ever be allowed to conflict?
- What does nesting do to all of this?

UNIVERSITY OF

ROCHESTER

COMPUTER SCIENCE

www.cs.rochester.edu/research/synchronization/