Transactional Memory Semantics
(Tracking Transactional State)

Michael L. Scott
University of Rochester
www.cs.rochester.edu/research/synchronization/
(EC)^2 Workshop, July 2008

with contributions from
Mike Spear, Virendra Marathe, Luke Dalessandro,
Sandhya Dwarkadas, and Arrvindh Shriraman
Transactional Memory

- Simplify synchronization for explicitly parallel programs
- Draw inspiration from the DB community
- Avoid standard problems with locks
  - obtain composability
  - avoid deadlock, priority inversion
  - eliminate clarity / concurrency tradeoff
  - (maybe) tolerate page faults & preemption; avoid convoying

```c
atomic {
    < your code here >
}
```

- Assumed implementation: speculation & rollback
A Privatization Puzzle

shared node* p

shared int n = 0;

A: atomic {
    my_node = p->next
    p->next = nil
    i = n
}
print i, my_node->val
delete my_node

B: atomic {
    if (p->next)
        p->next->val = 4
    n = 1
}

What might this code print?

» 0 3 (A first)  » 1 4 (B first)
» 1 3 ??  » 0 4 ??
» bus error ??
What’s Going On?

- Lack of agreement about how TM should behave
  - esp. if data can be accessed both inside and outside txns
  - nowhere near as easy as we once thought

- Lots of sticky issues
  - nesting
  - exceptions
  - condition synch. (retry)
  - irreversible ops / inevitable txns
  - progress guarantees
  - interaction w/ locks, NB data structures
  - ability to “leak” info from aborted txns
  - privatization and publication

source: slurmed.com
Outline

- Additional background
  - implementation realities
  - the publication / privatization problem
  - non-solutions
- Candidate semantics
  - serializability (database)
  - lock-based [Menon et al.]
  - ordering-based
- Distinguishing private use from error (time permitting)
- Conclusions / open questions
Starting Assumptions

- Transactions, at least in SW, may always have nontrivial OH
  - will we be willing to pay? (GC analogy?)
  - if not, “always use transactions” may be unacceptable (much as I like STM Haskell) — consider mesh app.
  - zero OH for private use (or close) may demand publication & privatization

- But SW txns serialize by reading & writing metadata that nontxnsal accesses ignore

- Resulting problems
  - delayed cleanup @ privatization
  - doomed txns @ privatization
  - early reads @ publication
Delayed Cleanup @ Privatization
[Marathe, '06; Larus & Rajwar, '07]

initially \( x = 0; \) \( x\_is\_public = true \)

B: atomic {
    if (x_is_public)
        x = 1
}

A: atomic {
    x_is_public = false
}

\( x = 0 \) ?? \hspace{1cm} // or \( x = 1 \) but B aborted?
Doomed Txns @ Privatization

[Wang et al., CGO'07; Spear et al., '07]

initially \( p == \&x; \) \( x\_is\_public == true \)

\[
\begin{align*}
B: \text{atomic} & \{ \\
& \text{if } (p\_is\_public) \{ \\
& \quad *p = 1 \\
& \}} \\
A: \text{atomic} & \{ \\
& \quad p\_is\_public = \text{false} \\
& \} \\
& p == \text{nil} \\
\end{align*}
\]
initially $x == 0$; $x_{\text{is\_public}} == false$

B: atomic {
    $a = 0$
    $t = \text{prefetch}(x)$
}

A: $x = 1$
    atomic {
        $x_{\text{is\_public}} = true$
    }
    if ($x_{\text{is\_public}}$) {
        $a = t$
    }
}
Non-Solutions

- Static data partitioning (unless cost is very low or perceived benefit is very high)
  - forces txnal OH on all accesses to shared data
  - doesn’t mix well with legacy code
  - has much of the complexity of the general case to enforce (don’t want to partition the type system)

- Strong isolation (nontxnal accesses serialize wrt transactions) [Blundell et al., CAL’06]
  - unclear what an access is at the language level
  - prevents compiler reordering of nontxnal accesses
  - is very expensive!
Issues for Publication/Privatization-Safe TM

- What are permissible programming idioms?
- How should correct programs behave?
- Is the language impl. required to catch bad programs? Statically?
- If not, are there constraints on what bad programs can do?

Policy choices
- txnal / nontxnal races are bugs
- consequences of bugs are limited — no “catch fire” semantics
  - in particular, no out-of-thin-air reads

NO  YES
Outline

- Additional background
  - implementation realities
  - the publication / privatization problem
  - non-solutions

- Candidate semantics
  - serializability (database)
  - lock-based [Menon et al.]
  - ordering-based

- Distinguishing private use from error (time permitting)

- Conclusions / open questions
Database Semantics

- **Serializability (S)**
  - Observed history must be equivalent to (same ops, same results) some serial history (no overlapping txns) with the same thread subhistories

- **Strict Serializability (SS)**
  - Additionally, if 2 txns (of different threads) do not overlap in the observed history, they must appear in the same order in the serial history
  - Motivation: prevent threads from using outside events to observe txns in the “wrong” order — plane ticket example
Single Lock Atomicity

- (SLA) Transactions behave “as if” they acquired a single global lock
  - Equivalent to SS:
    - serial txn order ≡ lock acquisition order
    - locks force order of wrt nontxnal accesses
  - Too expensive to implement
    - At begin_txn, must ensure no peer has prefetched published data
    - At end_txn, must ensure all previous txns have cleaned up, and all doomed txns aborted
Relaxing Order [Menon et al.’07]

- 3 progressive relaxations of transaction order
- ALA (asymmetric lock atomicity) the most appealing:
  txns behave “as if”
  » there is a separate reader-writer lock for every datum
  » we acquire read locks on all to-be-read data at begin_txn
  » we acquire write locks on to-be-written data at write time

Asymmetry reflects the fact that
» flow dependences are easier to catch than anti-
» nobody wants to publish by antidependence anyway
Problems w/ ALA, etc.

- Explains behavior in terms of (multiple) locks — which txns were supposed to replace!
- Abandons serial order for txns — arguably the key to success in the DB world
- Permits temporal loops (next slide)
Unorderable ALA Transactions

Initially \( T2_{\text{used}_x} = \text{false}; \ x = 0 \)

// T1                      // T2
C: atomic {
    t = prefetch(x)       // RL x
}                        // RL x, T2_{\text{used}_x}
A: x = 1
B: atomic {
    f = T2_{\text{used}_x}  // RL x, T2_{\text{used}_x}
    i = x
}
T2_{\text{used}_x} = \text{true}  // WL T2_{\text{used}_x}
    j = t
}

- at end:
  » A < B because \( a == 1 \)
  » B \('<'\) C because of anti-dep
  » C < A because \( j == 0 \)

- is B < C < B ok in a buggy program?
An Alternative Proposal

- Define semantics in terms of ordering (Cf: Java, C++)
- Keep transactions serial; make txnal-nontxnal ordering optional
- SSS (selective strict serializability)
  - transactions appear to occur in a serial order, $<_t$, consistent with program order, $<_p$
  - some txns are marked “publishing” &/or “privatizing”
    - may be either, both, or neither
  - global order, $<_g$, is an extension (superset) of $<_t$; also
    - if T1 is publishing and A $<_p$ T1 $<_t$ T2, then A $<_g$ T2
    - if T2 is privatizing and T1 $<_t$ T2 $<_p$ B, then T1 $<_g$ B
SSS (cont.)

- Read R is permitted to return the value of write W iff
  - R and W are unordered by \( <_g \) or
  - \( W <_p R \) or \( W <_g R \), and there is no intervening write
- If all txns are publishing and privatizing,
  \( SSS \equiv SS \equiv SLA \)
- If no txns are publishing or privatizing (e.g., w/ static data partitioning), \( SSS \equiv S \)

→ You pay for what you need
Unfortunately...

- Publishing txns still pay for safe publication by antidependence, which we don’t need
- SFS (selective flow serializability)
  - T1 < T2 if T2 (follows a txn that) reads a value T1 wrote
  - replace previous publication rule with
    - if T1 is publishing and A < T1 < T2, then A < g T2
- This is
  - cheaper than SSS
  - similar in spirit (but not equivalent) to ALA
    - racy programs subject to temporal loops
    - but correct programs pay for only what they need
Unorderable SFS Transactions

initially $T2\_used\_x == false; \ x == 0$

// T1                 // T2

C: atomic {
    t = prefetch(x)

A: $x = 1$
B: atomic {
    f = T2\_used\_x
    a = x
}

T2\_used\_x = true
b = t

• at end:
  » A < B because $a == 1$
  » B < C because of anti-dep
  » C ‘<’ A because $b == 0$

• is B < C < B ok in a buggy program?
Implementation

- We have publication / privatization-safe variants of
  - TL2
  - JudoSTM
  - RingSTM
  - “Sequence lock” (single orec) STM

- These achieve their safety in very different ways

- We are currently collecting data on the costs of (S)SS and (S)FS
Outline

● Additional background
  » implementation realities
  » the publication / privatization problem
  » non-solutions

● Candidate semantics
  » serializability (database)
  » lock-based [Menon et al.]
  » ordering-based

● Distinguishing private use from error (time permitting)

● Conclusions / open questions
Privatization v. Errors

- Policy: by default, nontxnal use of X is an error if X is ever accessed by more than one thread
- Require programmer to annotate private use
- But want to share code across contexts; can’t say
  \[ p = \text{private} \ p\rightarrow\text{next} \]
- Possible solution:
  \[ p = \text{private with next}* \text{head} \]
- Define rules for propagation of privateness
  » property of the reference, not the object
- Clone subroutines for realized combinations of private and shared args
Truly Private v. Sharable

- Performance issue, not correctness
- Property of the object, not the reference
- Can be deduced in many cases by the compiler
- Can be checked at run time OW
  - has cost, but often amortizable
- Can be refined to cover
  - private and local to txn
  - private and long-lived
  - sharable but not yet used in current txn
  - sharable and already read
  - sharable and already written
  - leaky?
Conclusions

- TM is not as easy as it looks (even to explain)
- Ordering-based semantics seem more promising than lock-based
  - positive experience from DB and language communities
  - (relatively) simple rules on readable values
- Selective enforcement of txnal-nontxnal ordering pays for publication/privatization only when needed
- Language and compiler integration of TM will be essential
Open Questions

- Is there a restricted version of the data race detection problem that would be tractable & suited to TM?
- Are there verification issues that should influence the choice of TM semantics?
- Are we on the right track with SFS?
- How do we verify that a TM implementation provides the chosen semantics?
  - ordering: A-C-I
  - progress: deadlock freedom, hopefully livelock freedom, ideally starvation freedom