Single Global Lock Semantics in a Weakly Atomic STM

Vijay Menon, Steve Balensiefer, Tatiana Shpeisman,
Ali-Reza Adl-Tabatabai, Rick Hudson,
Bratin Saha & Adam Welc

Intel Programming Systems Lab
Motivation

Is TM a semantic step back as a programming model?
- Transactions intended to replace lock-based critical sections
- STMs appear to “break rules” of lock-based code
- Replacing locks with atomic may provide unexpected results
- Non-transactional accesses are problematic

Strong vs. Weak
- Strong semantics are difficult to provide efficiently
- Most STMs are weakly atomic
- What are “safe” weak semantics?
- This talk: explore safe weak semantics from a Java perspective
### Example: Publication Idiom

Initially data = 42, ready = false, val = 0

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>data = 1</td>
<td>atomic { // transaction T2</td>
</tr>
<tr>
<td></td>
<td>if (ready)</td>
</tr>
<tr>
<td></td>
<td>val = data;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>Can val == 42?</td>
</tr>
</tbody>
</table>

Global variable **data** is accessed in & out of transaction

Program is still race-free under locks
- If T1 before T2 then val == 1
- If T2 before T1 then val is not written (i.e., val == 0)

This behavior is guaranteed by locks and STMs (Java, C++, ...).
Benign Modification?

Initially data = 42, ready = false, val = 0

Thread 1

data = 1
atomic { // transaction T1
    ready = true;
}

Thread 2

atomic { // transaction T2
    tmp = data;
    if (ready)
        val = tmp;
}

Can val == 42?

Access hoisted: compiler (speculative code motion) or STM (early copying)
- Apparent benign race introduced

With locks, we still expect same behavior as before:
- If T1 before T2 then val == 1; if T2 before T1 then val == 0 & tmp is dead
- val != 42 guaranteed by Java ... but, can break on most STMs
Publication Example in STM

Initially data = 42, ready = false, val = 0

Thread 1
1:
2:
3: data = 1
4: atomic { // transaction T1
5:   ready = true;
6: }
7:
8:
9:

Thread 2
atomic { // transaction T2
  tmp = data;

  if (ready)
    val = tmp; // val == 42
}

Most STMs can produce val == 42 because transactions can overlap
What is a correct execution?

Sequential consistency (Lamport ‘79)
- All threads agree on some total ordering
- Observable effects must be consistent that total ordering
- On single thread: allows standard compiler & hardware opts
- Multiple threads: much more limiting

Synchronization models (Adve/Hill ‘90)
- Define correctly synchronized programs
- Guarantee sequential consistency only for that subset

How does TM fit in?
Java’s Memory Model (JMM)

Data-race freedom == correctly synchronized

- Strong guarantee:
  - Sequentially consistent behavior for race-free programs

For racy programs

- Safety and security paramount
- No values “out-of-thin-air”
- “Happens-before” ordering must be obeyed
- JMM allows for benign races
Implications of JMM on STM

Preventing out-of-thin-air values

- Granular safety: no lost updates / inc. reads due to adjacent writes
- Observable consistency: no artificial races due to inconsistent execution
- Speculation safety: no visible speculative effects

Preserving happens-before ordering

- Privatization safety: order from txn access to non-txn access
- Publication safety: order from non-txn access to txn access

See paper for details ...
“Synchronization” Model for TM

Specifies what values can be seen by the reads
- Allows programmers to reason about their code
  - E.g, defines happens-before relations imposed by transactions
- Determines what compiler transformations are legal
- Sets the rules for TM implementation

Conflicting requirements

- Simplicity
  - Easy to use by non-expert programmer
- Flexibility
  - Allow for efficient TM implementation
Single Global Lock Atomicity (SGLA)

Transactions execute as if they are protected by a single global lock

```java
atomic {
    synchronized (global_lock) {
        S;
        S;
    }
}
```

Matches intuition of weakly atomic STM
- Transactions are serialized wrt each other
- Sequential consistency for race-free programs
- In Java, well-defined behavior for races

Has surprising consequences for TM implementations
Publication via Empty Transaction

Initially data = 42, ready = false, val = 0

Thread 1

\[
\begin{align*}
\text{data} &= 1 \quad // \text{S1} \\
\text{atomic \{ \} } & // \text{T1} \\
\text{ready} &= \text{true};
\end{align*}
\]

Thread 2

\[
\begin{align*}
\text{atomic \{ } & // \text{T2} \\
\text{tmp} &= \text{data}; \quad // \text{S2} \\
\text{if (ready)} & \\
\text{val} &= \text{tmp}; \\
\}
\]

Can \text{val} == 42?

If T1 before T2 then \text{val} == 0 or \text{val} == 1
If T1 after T2 then \text{ready} == false and \text{val} is not read
Under SGLA, empty transactions impose ordering constraints
Empty Transaction in STM

Initially data = 42, ready = false, val = 0

Thread 1
1:
2:
3:    data = 1       // S1
4:    atomic { }    // T1
5:    ready = true;
7:
8:
9:

Thread 2
atomic {                            // T2
    tmp = data;                     // S2

    if (ready)
        val = tmp;
}

Most STMs can produce val == 42 because transactions can overlap

Difficult to provide concurrency in STM under SGLA restriction
Disjoint Lock Atomicity (DLA)

Weaken SGLA: Only dynamically conflicting transactions execute as if they are protected by the same lock
- T1 and T2 conflict if both access location x and one writes

Same as single global lock atomicity for race-free programs

Eliminates unnecessary (?) ordering constraints in racy code
- Does not require handling publication via empty transaction

Less intuitive than single global lock atomicity
- Cannot statically construct an equivalent lock-based program
- As if locks are “magically” acquired at transaction start
Publication via Anti-Dependence

Initially data = 42, ready = false, val = 0

Thread 1

| data = 1 // S1 |
| atomic { // T1 |
| test = ready; |
|

Thread 2

| atomic { // T2 |
| tmp = data; // S2 |
| ready = true; |
| val = tmp; |
|

Can test == false and val == 42 ?

Under SGLA & DLA : No

- If T1 before T2 then test == false and val == 1
  - Anti-dependence from T1 to T2 through ready
- If T1 after T2 then test == true

Complication : “invisible” read in T1 not detectable by T2 in STM
Asymmetric Lock Atomicity (ALA)

Transactions execute as if every memory access is protected by a lock
- Read locks “magically” acquired at transaction start (like DLA)
- Write locks acquired lazily any time before first access (unlike DLA)
- All locks released at the end of transaction
- Provides sequential consistency for race-free programs
- Supports benign race in publication by flow dependence
Revisiting Pub. by Anti-dependence

Thread 1

```c
data = 1;
atomic { // T1
    // read lock L_{ready}
    ...
    test = ready;
} // release L_{ready}
```

Thread 2

```c
atomic { // T2
    // read lock L_{data}
    val = data
    // write lock L_{ready}
    ready = true;
    tmp = val
} // release L_{ready}, L_{data}
```

ALA does not support publication by anti-dependence

- No ordering between write of data (Thread 1) and read of data (Thread 2)
- Consequence of encounter-time write locking
- Easier to implement: T2 does not need be aware of read of ready in T1
  - Only needs to detect conflict with earlier writes
  - Invisible readers not problematic
Encounter-time Lock Atomicity (ELA)

Thread 1

```c
data = 1;
atomic { // T1
    ...
    // write lock L_ready
    ready = true;
} // release L_ready
```

Thread 2

```c
atomic { // T2
    // read lock L_data
    val = data
    // read lock L_ready
    if(ready)
        tmp = val
} // release L_ready, L_data
```

Relax ALA to acquire all locks lazily

- No "magic" - effectively models a pessimistic, encounter-time locking STM
- Still provides sequential consistency for race-free programs
- Less tolerant of benign races
  - Does not support racy publication via flow dependence
  - Programmer must avoid benign races
  - Restrictions on compiler optimization / STM implementation
- Still requires privatization safety (e.g., quiescence with invisible readers)
# Summary of Models

<table>
<thead>
<tr>
<th></th>
<th>Safe Publication by</th>
<th>Safe Priv.</th>
<th>Linearization / Ordering Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Empty Txn</td>
<td>Anti-Dep</td>
<td>Flow-Dep</td>
</tr>
<tr>
<td>SGLA</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>DLA</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>ALA</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>ELA</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Implementation: Write buffering (no out-of-thin air) + Linearization (ordering)  
- See paper for details
Traveling Sales Person Solver

Most time outside transactions (high strong atomicity overhead)
Program has a benign data race on current shortest path length
Single global lock atomicity same cost as unsafe write buffering
Strong atomicity is better for 16 threads
java.util.TreeMap

- Synthetic workload: 80% gets, 20% updates
- Virtually all time is spent in transactions

SGLA and DLA scale up to 4 threads, degrade quickly beyond
ALA and ELA level out beyond 4 threads
- Privatization safety (quiescence) is still expensive
Conclusion

Can provide safe weakly-atomic semantics ala JMM
- Single global lock atomicity possible, but expensive
- Can relax requirements for racy programs while still providing safety guarantees and tolerating benign races
- Global communication for privatization still a major performance challenge
- Overall performance not necessarily better than strong atomicity

Questions to consider:
- Is data race freedom the right model for correctness?
- What guarantees to make for racy / incorrectly synch. programs?
  - Native (C/C++)
  - Managed (Java/C#)
- Is there such a thing as a benign data race?
  - What do we tell programmers?
  - What do we tell compiler writers / STM implementers?
### Ordering: Privatization Safety

*Must respect happens-before ordering from transactional access S1 to conflicting non-transactional access S2*

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>`atomic { // T</td>
<td>[Privatizing action] // P2</td>
</tr>
<tr>
<td>S1;</td>
<td>S2;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Privatizing action**
- Any synchronization action that imposes inter-thread ordering
- Transaction, lock acquire, or volatile read
Privatization Example

Initially x != null, x.data = 0

Thread 1

atomic { // T
    if (x != null)
        x.data = 42; // S1
}

Thread 2

atomic { // P
    t = x;
    x = null;
}
val1 = x.data; // S2
val2 = x.data; // S2'

Can val1 == 42 and val2 == 0?

If P before T then S1 is never executed and val1 == val2 == 0
If P after T then val1 == val2 == 42

Requirement for race free programs
- Problem for early STMs - published solutions (CGO ’07, PLDI ’07)
Ordering: Publication Safety

*Must respect happens-before ordering from non-transactional access S1 to conflicting transactional access S2*

**Thread 1**

```
S1;
[Publishing action] // P1
```

**Thread 2**

```
atomic { // T2
    S2;
}
```

If P1 happens before T2 then S2 should observe the result of S1

Publishing action

- Any synchronization action that imposes inter-thread ordering
- Transaction, lock release or volatile write
Ordering: Publication Safety

STMs read data early before read & write sets are complete
- Speculative reads can see stale values in eager & lazy STMs

Not an issue for race free programs if:
- Programmer educated to avoid “benign” race
- Implementation cannot insert speculative loads in new program paths
  - Avoid speculative code motion for loads in compiler
  - Avoid early reads in “shadow copies” in STM
    (Granular Inconsistent Read)
  - Otherwise, may introduce “benign” race

Racy programs
- Managed code: must still respect ordering rules
- Native code: catch fire semantics allows us to ignore
No-Thin-Air: Granularity Safety

No observable writes to fields not accessed in a transaction

- Important for race free programs

Initially \( x.g = 0 \)

Thread 1

1: \[
\text{atomic} \\
\text{x.f = 1; // buffer \{1, 0\}}
\]

3: 

4: \[
\text{}/\text{ Commit: x.f = 1, x.g = 0}
\]

5: 

Thread 2

3:

4: \[
x.g = 1;
\]

\( x.g == 0 \) cannot occur with locks

\( x.g == 0 \) can occur in STM due to coarse granularity of version management (Granular Lost Update)
No-Thin-Air: Observable Consistency

Side effects may only be visible if they are based on a consistent state of memory

Known issue for native code
- Potentially faulting instructions -> catastrophic failure
- Previously considered non-issue for managed code - exceptions can be lazily validated
  Initially $x == y == 0$

```
Thread 1
1:    atomic {
2:      // open read x, y
3:      x++; // fault!
4:    }
5:    y++; // fault!
6:  }

Thread 2
atomic {
  // open read x, y
  if (x != y)
    *0; // fault!
}  ```
Observable Consistency for Managed Code

Initially $x = y = z = 0$

Inconsistent writes are still a problem!

With locks this program is race-free: $z = 1$ is never executed

**An STM that does not preserve consistency may introduce a race**
No-Thin-Air: Speculation Safety

Even in a consistent state, speculative values should not be visible to other threads

Initially $x = y = z = 0$

Thread 1

1: atomic {
2: 
3: 
4: 
5: 
6: 
7:  
8:  
9:  
10: }

Thread 2

atomic {
    if ($x == 0$)
        $y = 1$;
    else
        $z = 1$;
/* abort */
}

Thread 3

t = $y$;

Can $z == 1$ and $t == 1$?
No-Thin-Air: Speculation Safety

An in-place update STM can produce results inconsistent with any single execution.

- Mixes results of different, incompatible executions
- Speculative value appears out-of-thin-air
  - Speculative Lost Update
  - Speculative Dirty Read
- Problems arise due to race in original program
  - A race allows another thread to see speculative state
- Not a problem for race free programs
<table>
<thead>
<tr>
<th></th>
<th>Segregated Programs Only</th>
<th>Race Free Programs Only</th>
<th>All Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privatization Safety</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Publication Safety</td>
<td>No</td>
<td>No *</td>
<td>Managed</td>
</tr>
<tr>
<td>Granular Safety</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Observable Consistency</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Speculation Safety</td>
<td>No</td>
<td>No</td>
<td>Managed</td>
</tr>
</tbody>
</table>
Implementation Overview

Baseline implementation: write-buffering
- Optimistic concurrency for reads
- Encounter-time locking for writes
- Commit log for buffered writes

Write buffering with commit log -> no values out-of-thin-air
- Speculation safety: speculative state is not visible
- Granularity safety: no lost update / inc. read due to adjacent write

Insufficient to preserve ordering constraints
- Publication safety: ordering from non-txn access to txn access
- Privatization safety: ordering from txn access to non-txn access
Total Linearization for SGLA

Invariant: $T_{1\text{\_start}} < T_{2\text{\_start}} \Rightarrow T_{1\text{\_val}} < T_{2\text{\_val}} \Rightarrow T_{1\text{\_end}} < T_{2\text{\_end}}$

- Publication safety: $T_1 < T_2 \Rightarrow S_1 < T_2$
- Privatization safety: $T_1 < T_2 \Rightarrow T_1 < S_2$

Expensive to implement: global synch., only permits pipelined concurrency
Decoupled Linearization for DLA

Same as total linearization if T1 and T2 conflict
Allows nested overlap if T1 and T2 do not conflict
Still expensive - invisible readers requires conservative solution
Lazy Start Linearization for ALA

Enforces start linearization only if T2 writes x later read by T1
- Lazily detects violations of serialization
- Uses timestamps instead of version numbers to detect
- TL2 already subsumes this check for consistency (Dice/Shalev/Shavit '06)
Experimental Evaluation

Platform:
- 16-way 2.2 GHz Xeon® comprised of 4 boards with 4 processors each
- 8KB L1, 512KB L2, 2MB L3 cache per processor
- Shared 64MB L4 cache per board

Benchmarks:

TSP: multi-threaded Traveling Salesman solver
- Small transactions (fine-grain locking)
- Most of time is spent outside of transactions
- Benign race on current short path

TreeMap: red-black tree from Java class library
- Synthetic workload: 80% gets, 20% updates
- Virtually all time is spent in transactions
Racy publication

Initially data = 42, ready = false, val = 0

Thread 1

```java
data = 1

atomic { // transaction T1
    ready = true;
}
```

Thread 2

```java
atomic { // transaction T2
    tmp = data;
    if (ready)
        val = tmp;
}
Can val == 42?
```

Race on data even under locks (when T2 ->hb T1)
Appears benign (tmp is dead in this ordering)
JMM still guarantees correctness in lock-based variant