Designing Transactional Memory Systems

Part II: Obstruction-free STMs

Pascal Felber
University of Neuchatel
Pascal.Felber@unine.ch

Institut d'informatique
Université de Neuchâtel

Based on joint work with Christof Fetzer & Torvald Riegel
with slides borrowed from several other people

Agenda

- **Part I:** Introduction
- **Part II:** Obstruction-free STMs
  - DSTM: an obstruction-free STM design
  - FSTM: a lock-free STM design
  - LSA-STM: a time-based STM design
- **Part III:** Lock-based STMs

Why obstruction freedom?

Obstruction freedom **"Any thread that runs by itself for long enough makes progress"**

vs.

Lock freedom **"Some thread always makes progress"**

- Obstruction freedom argued to be strong enough in practice
- Obstruction freedom easier to implement efficiently than lock freedom

DSTM [Herlihy et al., 2003]

- First **dynamic** STM
  - No need to know which data will be accessed a priori
  - Object-based, Java implementation
  - Non-blocking (obstruction free)
- Simple API

Wrapper around ordinary object

```
void    beginTransaction();
Object  open(TMObject obj, READ|WRITE);
boolean commitTransaction();
```

DSTM: principle

- **Problem:** update a set of objects atomically
- **Solution:**
  - Objects accessed indirectly through "locators"
  - Transaction state (active, committed, aborted) can be read/updated by other transaction
  - Objects must be **opened** before use
  - Objects opened in write mode are only acquired, updates are local until commit
  - Reads are essentially invisible
  - Incremental validation for consistent reads

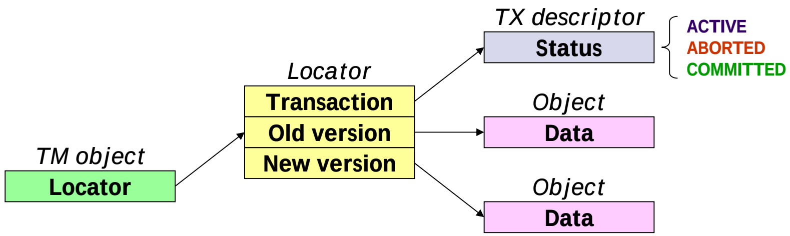
Why consistent reads?

- Although no "damage" is done to shared data (consistent writes), inconsistent reads can create program crashes, infinite loops, etc.

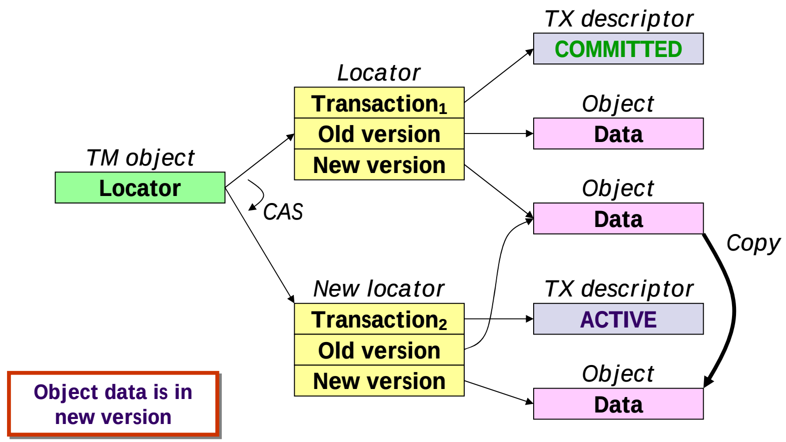
```
// Invariant: x + y == 0
// Initially: x = y = 0
                                START;
                                a = x;              // 0
START;
a = x;              // 0
b = y;              // 0
assert(a + b == 0);
x = a + 1;          // 1
y = b - 1;          // -1
COMMIT;
// Here: x == 1 && y == -1
                                b = y;              // -1
                                assert(a + b == 0); // Ooops!
```


DSTM: data structures

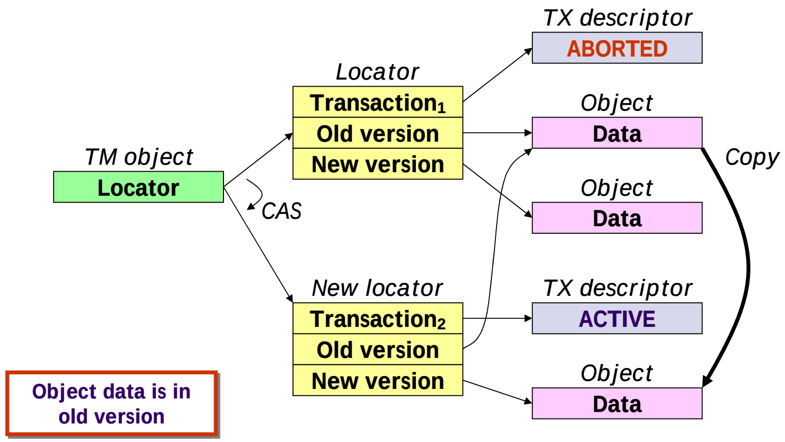
- Transaction acquires a free object (while opening it) by registering its locator
- Object is free if it does **not** contain the locator of an **active** transaction
- Locator holds two object versions (old, new)



DSTM: open after commit



DSTM: open after abort



DSTM: conflict management

- Conflicts are detected by checking status of owner transaction when opening object
- Conflicts are handled by a **contention manager** (CM)
  - Decide which transaction to kill, delay, or let go
  - To kill a transaction, CAS its status to ABORTED
  - CM is an independent component (one can register custom CMs)
  - Choosing the right contention manager is crucial to system throughput

DSTM: validation, commit, abort

- Validation is necessary on open
  - Check that read versions are still latest
  - Check that status is still **ACTIVE**
- Commit requires two phases
  - Validate read set
  - CAS state to **COMMITTED** (atomically update all objects opened in write mode)
- Transaction can also abort
  - CAS state to **ABORTED** (atomically release all objects opened in write mode)

DSTM: obstruction free

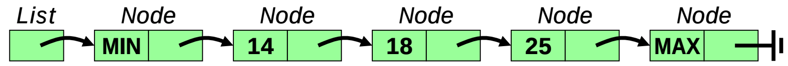
"Any thread that runs by itself for long enough makes progress"

- A transaction *T* can unilaterally abort other transactions
- Hence, *T* running on its own, can eventually commit

DSTM: costs

- Given *W* objects opened in write mode and *R* in read mode
  - *W + 1* CAS
  - *W* cloning overhead
  - *O((R + W) R)* validation overhead

DSTM: programming example



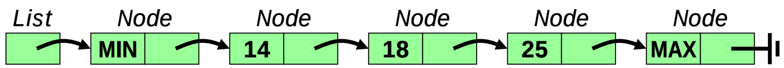
Non-transactional

```
public class Node {
 private int value;
 private Node next;

 public Node(int v) { value = v; }

 public void setValue(int v) { value = v; }
 public void setNext(Node n) { next = n; }

 public int getValue() { return value; }
 public Node getNext() { return next; }
}
```

Transactional

```
public class Node implements TMCloneable {
 private TMObject next;

 public void setNext(TMObject n) { … }
 public TMObject getNext() { … }

 public Object clone() {
  Node n = new Node(value);
  n.next = next;
  return n;
 }
 …
}
```


DSTM: programming example



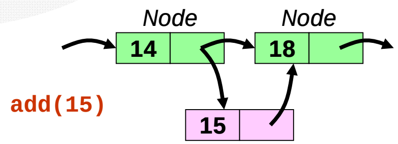
Non-transactional

```
public class List {
 private Node head;

 public List() {
  Node min = new Node(Integer.MIN_VALUE);
  Node max = new Node(Integer.MAX_VALUE);
  min.setNext(max);
  head = min;
 }
 // …
}
```

Transactional

```
public class List {
 private TMObject head;

 public List() {
  Node min = new Node(Integer.MIN_VALUE);
  Node max = new Node(Integer.MAX_VALUE);
  min.setNext(new TMObject(max));
  head = new TMObject(min);
 }
 // …
}
```


DSTM: programming example



Non-transactional

```
public boolean add(int v) {
 Node prev = head;
 Node next = prev.getNext();
 while (next.getValue() < v) {
  prev = next;
  next = prev.getNext();
 }
 if (next.getValue() == v)
  return false;
 Node n = new Node(v);
 n.setNext(prev.getNext());
 prev.setNext(n);
 return true;
}
```

Transactional

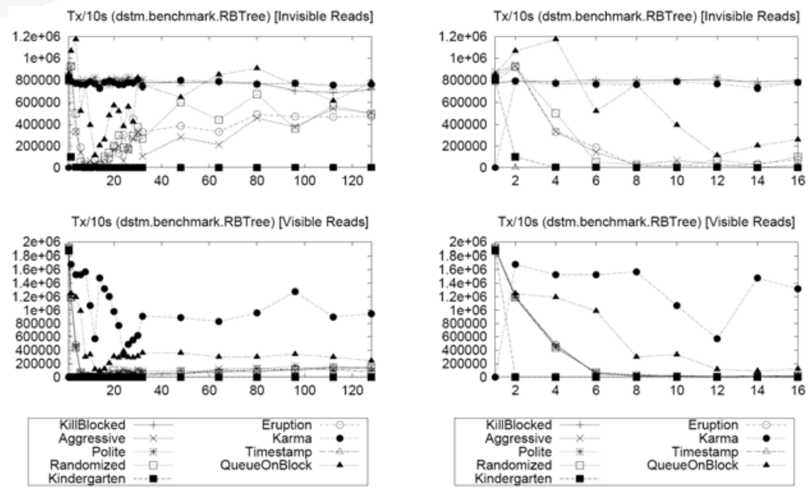
```
public boolean add(int v) {
TMThread t =
  (TMThread)Thread.currentThread();
 while (true) {
  t.beginTransaction();
  boolean result = false;
  try {
   Node prev = (Node)head.open(READ);
   Node next =
    (Node)prev.getNext().open(READ);
   while (next.getValue() < v) {
    prev = next;
    next = (Node)prev.getNext().open(READ);
   }
   if (curr.getValue() != v) {
    result = true;
    n.setNext(prev.getNext());
    prev = (Node)prev.open(WRITE);
    prev.setNext(new TMObject(new Node(v)));
   }
  } catch (Denied d) {}
  if (t.commitTransaction())
   return result;
 }
}
```

CM: how important?

- CM is essential for performance and livelock avoidance
- Sample CMs
  - Aggressive: kill enemy
  - Polite: exponential backoff first
  - Karma: increase priority with opened objects and retries, higher priority wins
  - Timestamp: older transaction wins
  - Greedy: uses timestamp-based priorities, bounds on worst case completion time

```
// Aggressive CM
void handleConflict(TX me, TX enemy) {
  enemy.abort();
}
```


CM: how important?



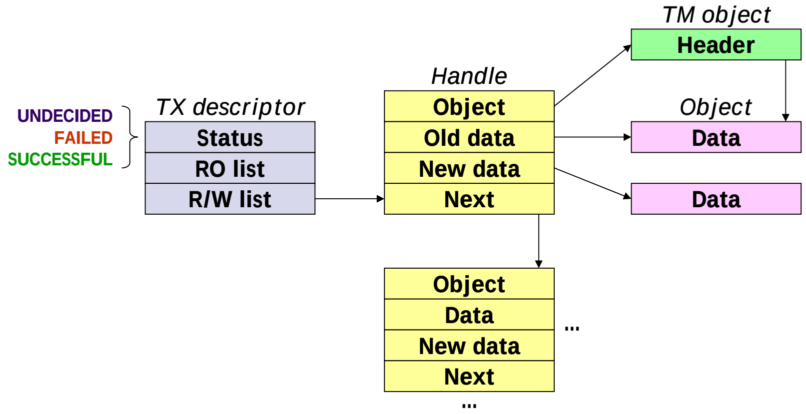
SXM [Herlihy, 2005]

- As DSTM, but:
  - C# implementation
  - Use visible reads (maintain reader list)
    - Single writer or multiple readers allowed
  - Support for some advanced patterns
    - **Conditional waiting** (retry when some object accessed by transaction have been updated)
    - **Or-else combinator** (specify alternative to use upon retry)

FSTM [Fraser, 2003]

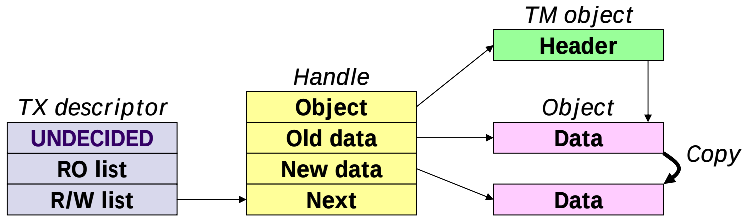
- Provides lock freedom (stronger than obstruction freedom!)
  - Implemented using helping (a transaction can help another one)
- Uses invisible reads
- No extra indirection (i.e., faster data access)
- Acquire objects at commit time (lazy)

FSTM: data structures



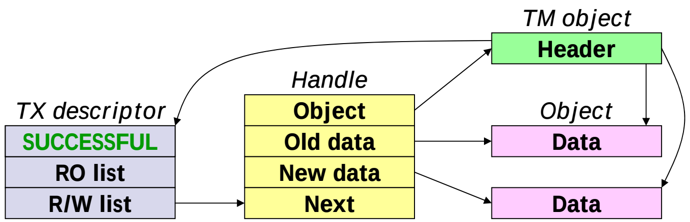
FSTM: open (write mode)

- Create shadow copy (to be updated) and store object in R/W list
- Note that the write is not visible to other transactions at this point



FSTM: commit

1. Acquire the objects in some total order
   - Header points to the transaction descriptor
2. Decision (after RO list validation)
3. Release objects
   - Header points to new copy



FSTM: lock freedom

- Commit phase ≡ multi-word CAS
  - Objects are acquired in some total order to ensure lock freedom
  - Contention is detected when the header points to another transaction
  - Contention is resolved by order based "helping"
    - If header points to the descriptor of another transaction, recursively help it complete
  - Conflict detected if old versions have changed

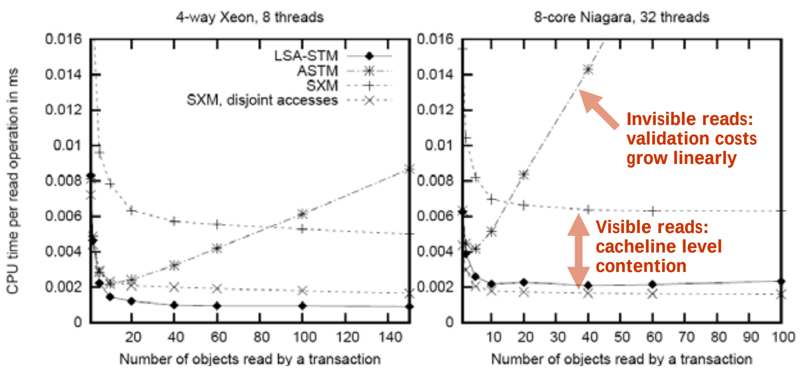
FSTM: costs

- Given *W* objects opened in write mode and *R* in read mode
  - *2W + 1* CAS
  - *W* cloning overhead
  - *O(R)* validation overhead (but may work on inconsistent data!)

On STM read operations

- Visible reads
  - Maintain reader list per transactional object
  - Can be used to detect R/W conflicts (pessimistic)
  - Contention on reader lists (e.g., root of tree)
- Invisible reads
  - No list of readers is maintained (optimistic)
  - No easy way to detect R/W conflicts
  - Consistency must be checked (validation)
    - **Validate on commit:** may work on inconsistent data
    - **Validate on open:** costly (linear w/ read set size)
- **Goal:** low validation costs + consistency

On the cost of read operations



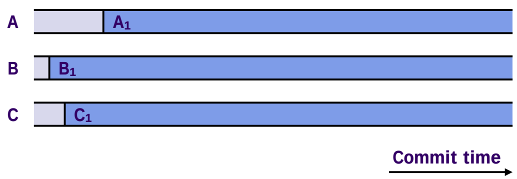
ASTM: invisible reads, eager validation
SXM: visible reads
LSA: time-based invisible reads

LSA-STM [Riegel et al., 2006]

- Motivation
  - Speed up for transactions with large read sets
  - Efficient time-based snapshot algorithm (LSA) to reduce overhead
- Read-only transactions
  - Keep multiple object versions (no abort)
- LSA-STM
  - Object-based (uses DSTM-like locators)
  - Java implementation
  - Annotations and AOP for ease of use
  - Winner of SUN's CoolThreads contest!

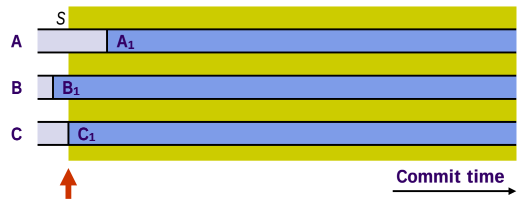
LSA-STM: algorithm

- Global time base: *CT*
  - Counts the number of commits
- STM objects have multiple versions
  - Each version *V* has a validity range R_V w.r.t. *CT*
  - Most recent version has upper bound ∞



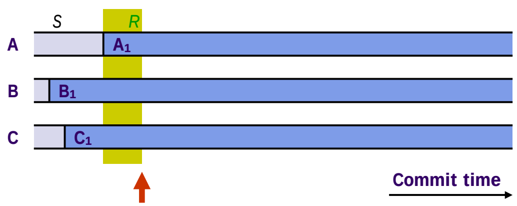
LSA-STM: algorithm

- Transaction maintains a “snapshot” with a validity range R_T
  - Equal to the intersection of the accessed versions' validity ranges
  - Initialized to $[S_T,\infty]$
  - If it becomes empty, transaction must abort



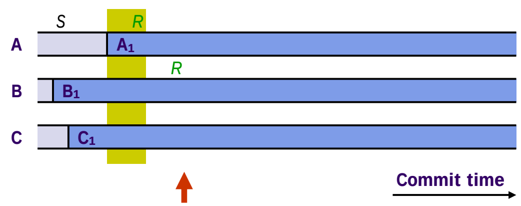
LSA-STM: algorithm

- Upon read, snapshot is updated
  - Validity range ends at time of the read
  - We know that the value read is valid now, but we don’t know if it will change in the future



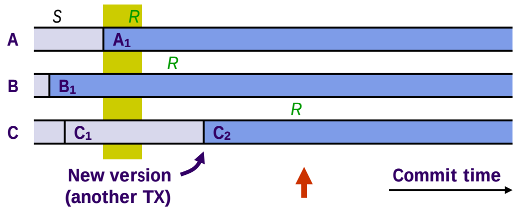
LSA-STM: algorithm

- Upon read, if snapshot intersects with the latest version’s validity range:
  - The snapshot is a valid linearization point (as long as there are no writes)
  - No need to update snapshot



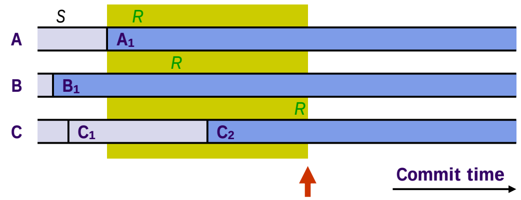
LSA-STM: algorithm

- Upon read, if snapshot **does not** intersect with the latest version's validity range:
  - The snapshot is a not valid linearization point
  - Must try to "extend" snapshot (may fail)
- Note: read-only transactions can use old version



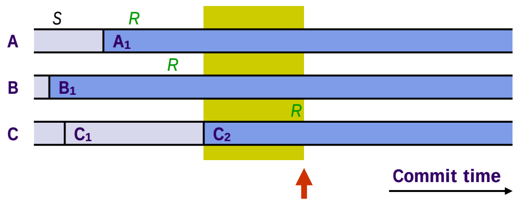
LSA-STM: algorithm

- Extension tries to increase the upper bound of the snapshot
  - Check if all versions read are still valid
  - If so, we can extend the upper bound of the snapshot to current *CT* (now)



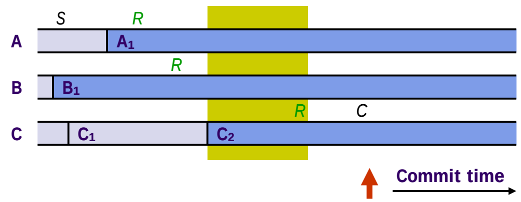
LSA-STM: algorithm

- Extension may also increase the lower bound of the snapshot
  - Set to the largest lower bound among the validity ranges of accessed versions



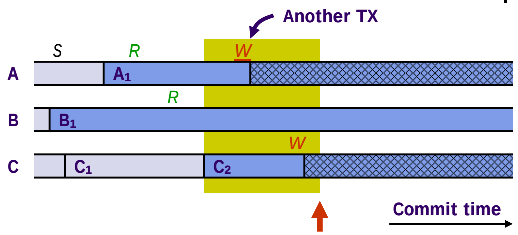
LSA-STM: algorithm

- Read-only transactions can commit as long as their snapshot is not empty
  - No need to extend range to current *CT*
  - Linearization point anywhere in snapshot range



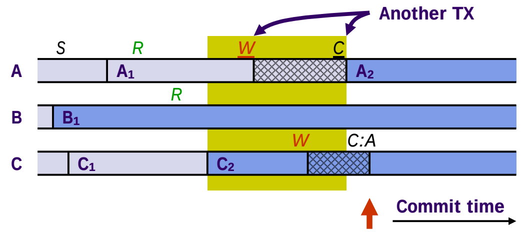
LSA-STM: algorithm

- Update transactions create new versions of modified objects upon commit at C_T
  - Validity range of newly created object versions starts at C_T
  - Tentative versions being written are not visible to other transactions and are discarded upon abort



LSA-STM: algorithm

- Upon commit, an update transactions tries to acquire a new, unique commit timestamp C_T
  - Transaction can commit iff the snapshot can be extended to C_T - *1* (otherwise, abort)
  - Note: validation can be skipped if $S_T = C_T - 1$



LSA-STM: algorithm [DISC 2006]

```
procedure START(T)                                   ▷ Initialize transaction attributes
    T.min ← CT                                       ▷ = min(R'_T)
    T.max ← ∞                                        ▷ = max(R'_T)
    T.O ← ∅                                          ▷ Set of objects accessed by T
    T.open ← true                                    ▷ Can T still be extended?
    T.update ← false                                 ▷ Is T an update transaction?
end procedure

procedure OPEN(T, o_i, m)                            ▷ T opens o_i in mode m (read or write)
    if m = write then
        T.update ← true
    end if
    if ⌊o_i^CT⌋ > T.max then                         ▷ Is most recent version too recent?
        if T.update ∧ T.open then                    ▷ Try to extend?
            EXTEND(T)
        end if
    end if
    if ⌊o_i^CT⌋ ≤ T.max then                         ▷ Can we use the latest version?
        T.min ← max(T.min, ⌊o_i^CT⌋)                 ▷ Yes, T remains open if it is still open
        T.max ← min(T.max, CT)
    else if ¬T.update ∧ VersionAvailable(o_i^T.max) then
        T.open ← false                               ▷ No, T.max has reached its maximum
        T.min ← max(T.min, ⌊o_i^T.max⌋)
        T.max ← min(T.max, ⌈o_i^T.max⌉)
    else                                             ▷ Cannot maintain snapshot
        ABORT(T)
    end if
    T.O ← T.O ∪ {o_i}                                ▷ Access object
end procedure
```

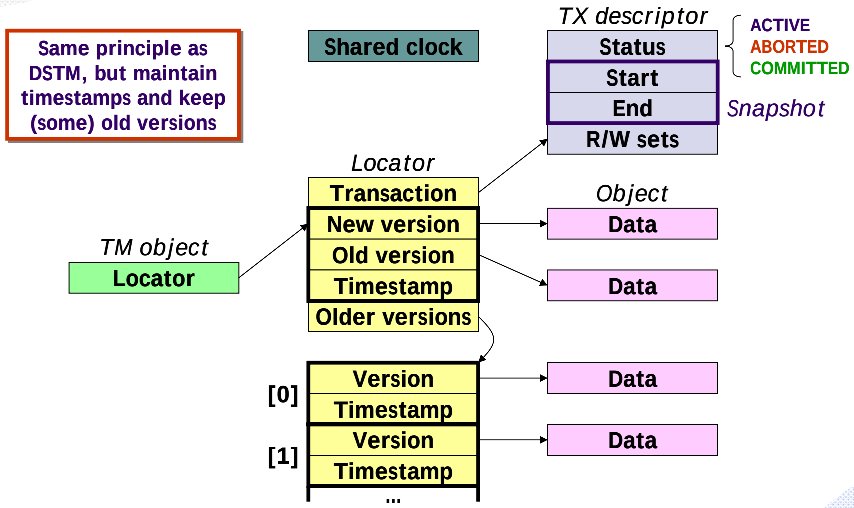

LSA-STM: algorithm [DISC 2006]

```
procedure EXTEND(T)                                  ▷ Try to extend the validity range of T
    T.max ← CT
    for all o_i ∈ T.O do                             ▷ Recompute the whole validity range
        T.max ← min(T.max, max(R'_{i,*}))
    end for
    if T.max < CT ∧ T.update then
        ABORT(T)                                     ▷ Update transaction must access most recent versions
    end if
end procedure

procedure COMMIT(T)                                  ▷ Try to commit transaction
    if T.update then
        CT_T ← (CT ← CT + 1)                         ▷ Acquire T's unique commit time CT_T
        if T.max < CT_T − 1 then
            EXTEND(T)                                ▷ For update transactions, CT_T and R'_T must overlap
            if T.max < CT_T − 1 then
                ABORT(T)
            end if
        end if
    end if                                           ▷ T can now be safely committed
end procedure
```

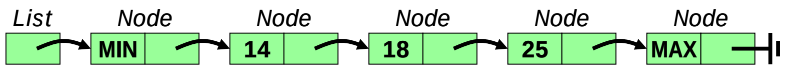
LSA-STM: # extensions required

- **Read-only transactions**
  - 0 (if enough versions are kept)
- **Update transactions**
  - 0 or 1 for commit
- At most one extension per accessed object
  - Only caused by concurrent updates to these objects
  - Disjoint updates do not increase the number of extensions
- In practice, only a few extensions are required

LSA-STM: data structures



LSA-STM: programming example



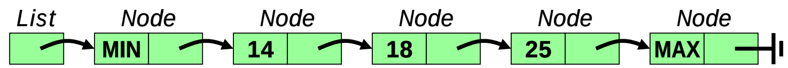
Non-transactional

```
public class Node {
 private int value;
 private Node next;

 public Node(int v) { value = v; }

 public void setValue(int v) { value = v; }
 public void setNext(Node n) { next = n; }

 public int getValue() { return value; }
 public Node getNext() { return next; }
}
```

Transactional

```
@Transactional
public class Node {
 private int value;
 private Node next;

 public Node(int v) { value = v; }

 public void setValue(int v) { value = v; }
 public void setNext(Node n) { next = n; }

 @ReadOnly
 public int getValue() { return value; }
 @ReadOnly
 public Node getNext() { return next; }
}
```


LSA-STM: programming example



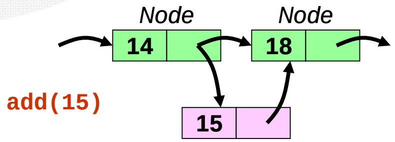
Non-transactional

```
public class List {
 private Node head;

 public List() {
  Node min = new Node(Integer.MIN_VALUE);
  Node max = new Node(Integer.MAX_VALUE);
  min.setNext(max);
  head = min;
 }
 // …
}
```

Transactional

```
public class List {
 private Node head;

 public List() {
  Node min = new Node(Integer.MIN_VALUE);
  Node max = new Node(Integer.MAX_VALUE);
  min.setNext(max);
  head = min;
 }
 // …
}
```

LSA-STM: programming example



Just add annotations to transactional objects and atomic methods… *et voilà !*

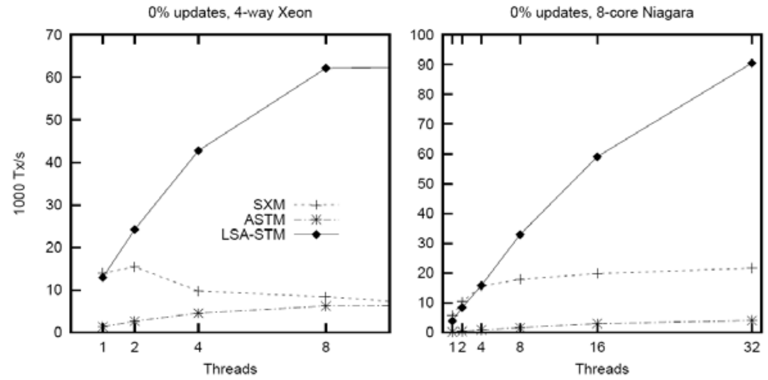
Non-transactional

```
public boolean add(int v) {
 Node prev = head;
 Node next = prev.getNext();
 while (next.getValue() < v) {
  prev = next;
  next = prev.getNext();
 }
 if (next.getValue() == v)
  return false;
 Node n = new Node(v);
 n.setNext(prev.getNext());
 prev.setNext(n);
 return true;
}
```

Transactional

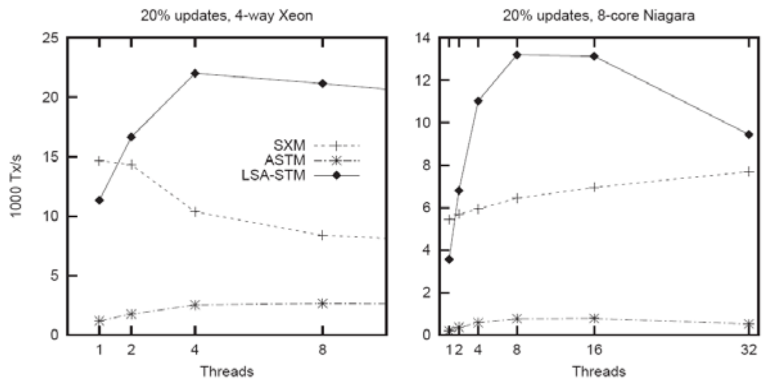
```
@Atomic
public boolean add(int v) {
 Node prev = head;
 Node next = prev.getNext();
 while (next.getValue() < v) {
  prev = next;
  next = prev.getNext();
 }
 if (next.getValue() == v)
  return false;
 Node n = new Node(v);
 n.setNext(prev.getNext());
 prev.setNext(n);
 return true;
}
```


LSA-STM: Linked list



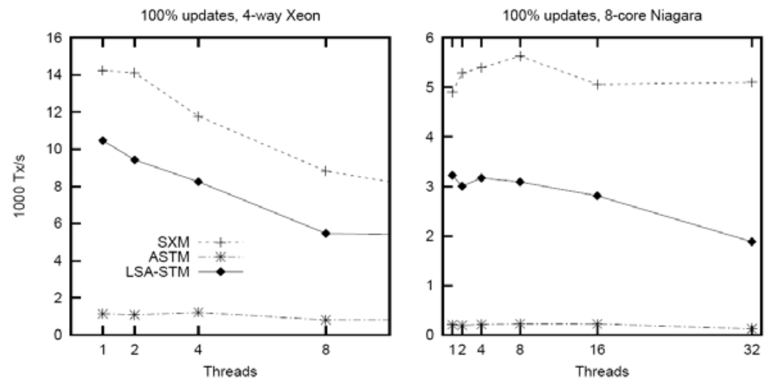
ASTM: invisible reads, eager validation
SXM: visible reads
LSA: time-based invisible reads

LSA-STM: Linked list



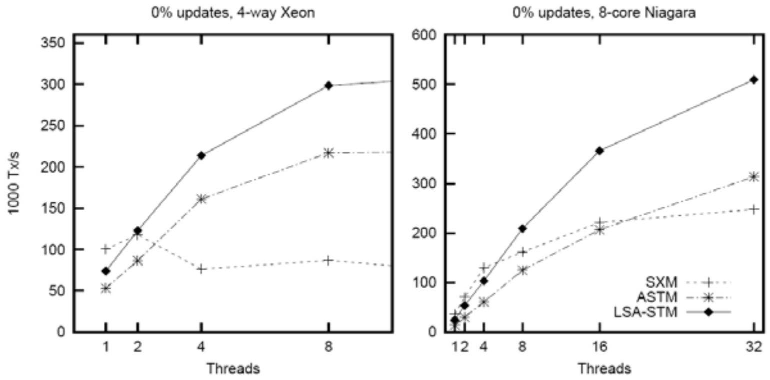
ASTM: invisible reads, eager validation
SXM: visible reads
LSA: time-based invisible reads

LSA-STM: Linked list



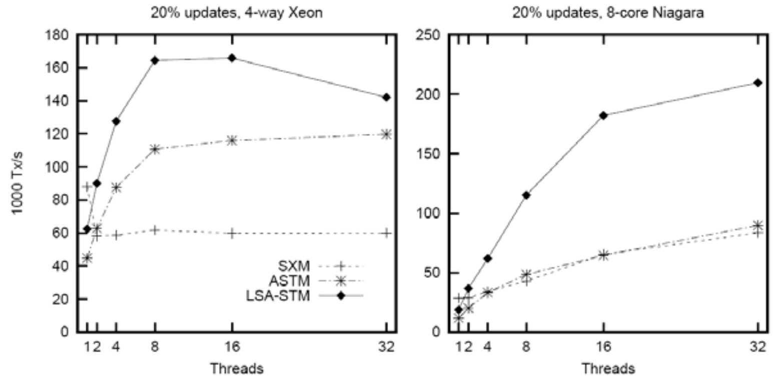
ASTM: invisible reads, eager validation
SXM: visible reads
LSA: time-based invisible reads

LSA-STM: Skip list



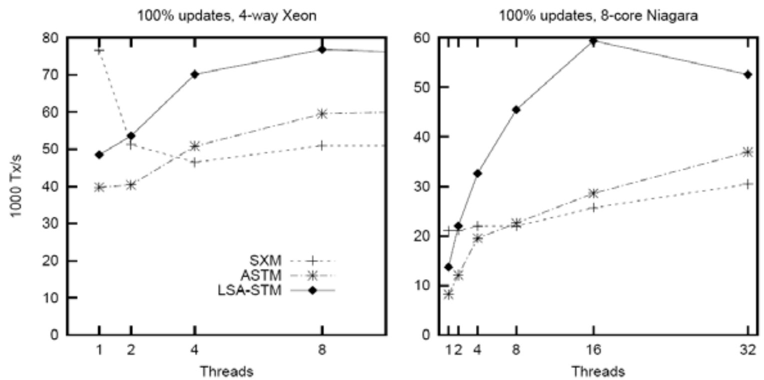
ASTM: invisible reads, eager validation
SXM: visible reads
LSA: time-based invisible reads

LSA-STM: Skip list



ASTM: invisible reads, eager validation
SXM: visible reads
LSA: time-based invisible reads

LSA-STM: Skip list



ASTM: invisible reads, eager validation
SXM: visible reads
LSA: time-based invisible reads

Conclusion (Part II)

- **Obstruction-free** STM designs provide progress guarantees (with the help of CM)
  - Transactions must be able to commit atomically

  … and abort another transaction atomically
  - Typically use indirection (must be able to "steal" objects)
- **Lock-free** is more complex to implement
  - Typically based on helping
- **Time-based** designs with invisible reads provide high efficiency and consistency
  - May be obstruction-free (or not…)