A Lazy Snapshot Algorithm with Eager Validation

Torvald Riegel[1] Pascal Felber[2] Christof Fetzer[1]

[1]Dresden University of Technology, Germany
{torvald.riegel,christof.fetzer}@inf.tu-dresden.de

[2]University of Neuchâtel, Switzerland
pascal.felber@unine.ch

DISC 2006

Motivation: Transactional Memories

1. Transactions with larger read sets
   $\longrightarrow$ Efficient snapshot algorithm to reduce the read overhead
2. Read-only transactions
   $\longrightarrow$ Multiple object versions
3. Variety of systems
   $\longrightarrow$ Target CMT, SMP, ...

Results: Lazy Snapshot Algorithm (*LSA*) and LSA-STM

Read Overhead

Visible reads:

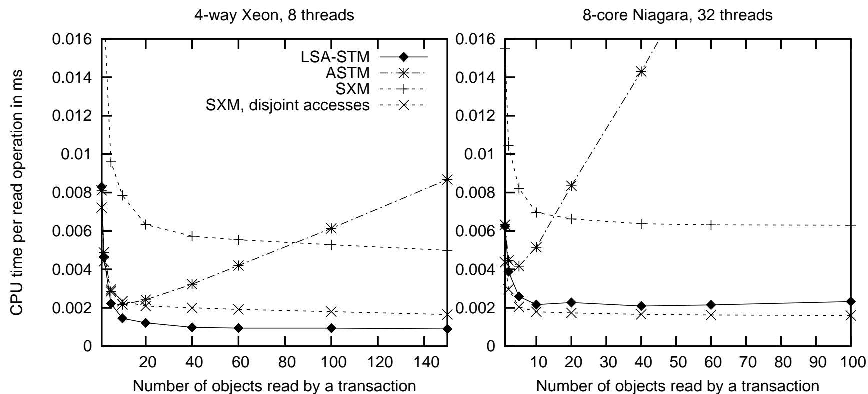
- List of readers $\rightarrow$ update memory $\rightarrow$ expensive, contention

Invisible reads:

- Optimistic $\rightarrow$ consistency of accessed data has to be checked (validate read set: check *every* already read object for changes)
- Only validate on commit $\rightarrow$ operate on inconsistent data
- Validate when read set grows $\rightarrow$ high validation costs

Goal: avoid validation *and* guarantee consistency

Read Overhead (2)



object-based STMs with Java implementations
ASTM/SXM: prototypes based on Marathe *et al.* / Guerraoui *et al.*, DISC 2005

Why validation?

Working with inconsistent data can influence:

- Termination
- Resource allocation/usage
  - No isolation (e.g., malloc)
- Exceptions, faults
  - Are false alarms acceptable?
  - How expensive is recovery?

Inconsistent data makes runtime behaviour less predictable

When to validate?

Possible optimizations:

- Explicit (programmer-controlled) validation: too difficult
- Based on transaction semantics (ensure termination)?
- Periodically?
  - Which validation period?
    → Did the transaction work on inconsistent data before we tried to validate?
- Before everything that fails if data is inconsistent?
  - Plenty of checks required → lots of validations

Problems: Complex runtime support, libraries

LSA: snapshot is always consistent
→ recompute only if the snapshot must be valid for a longer time

Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

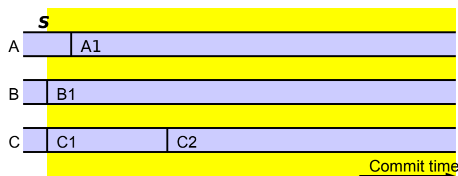
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

Snapshot of a transaction T has a validity range R_T

- Equal to the intersection of the accessed versions' validity ranges
- Initialized to $[start_T,]$
- Validity of a most recent version ends at time of the read

Transactions can try to *extend* R_T if necessary:

- Validity ranges are recomputed $\longrightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

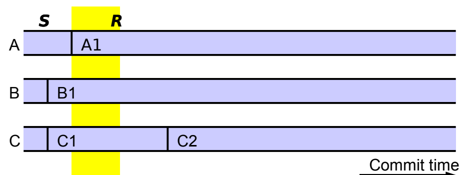
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

Snapshot of a transaction T has a validity range R_T

- Equal to the intersection of the accessed versions' validity ranges
- Initialized to $[start_T,]$
- Validity of a most recent version ends at time of the read

Transactions can try to *extend* R_T if necessary:

- Validity ranges are recomputed $\longrightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

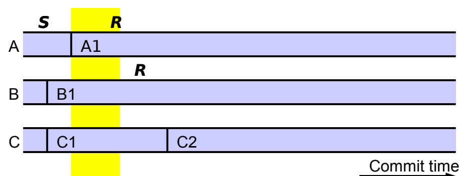
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

Snapshot of a transaction T has a validity range R_T

- Equal to the intersection of the accessed versions' validity ranges
- Initialized to $[start_T,]$
- Validity of a most recent version ends at time of the read

Transactions can try to *extend* R_T if necessary:

- Validity ranges are recomputed $\rightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

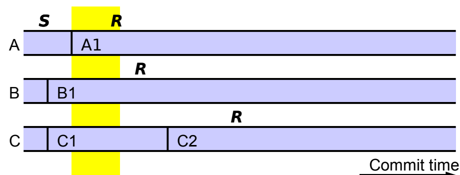
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

Snapshot of a transaction T has a validity range R_T

- Equal to the intersection of the accessed versions' validity ranges
- Initialized to $[start_T,]$
- Validity of a most recent version ends at time of the read

Transactions can try to *extend* R_T if necessary:

- Validity ranges are recomputed $\longrightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

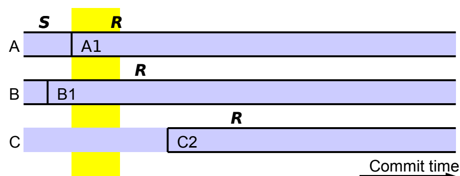
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

Snapshot of a transaction T has a validity range R_T

- Equal to the intersection of the accessed versions' validity ranges
- Initialized to $[start_T,]$
- Validity of a most recent version ends at time of the read

Transactions can try to *extend* R_T if necessary:

- Validity ranges are recomputed $\longrightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

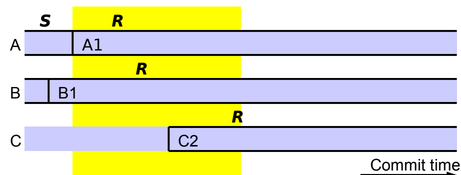
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

Snapshot of a transaction T has a validity range R_T

- Equal to the intersection of the accessed versions' validity ranges
- Initialized to $[start_T,]$
- Validity of a most recent version ends at time of the read

Transactions can try to *extend* R_T if necessary:

- Validity ranges are recomputed $\longrightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

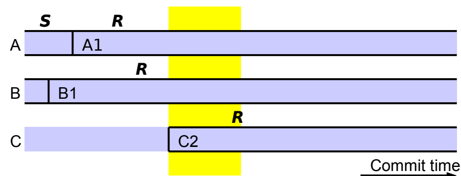
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

Snapshot of a transaction T has a validity range R_T

- Equal to the intersection of the accessed versions' validity ranges
- Initialized to $[start_T,]$
- Validity of a most recent version ends at time of the read

Transactions can try to *extend* R_T if necessary:

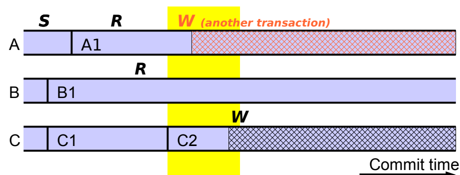
- Validity ranges are recomputed $\rightarrow$ possibly larger upper bound for R_T



Timestamp-based Transactions

Update transactions T:

- Updating an object creates a new, most recent version
- On commit, unique commit timestamp CT_T is acquired
- Transaction can commit iff R_T can be extended to $CT_T - 1$
- Validity of newly created object versions starts at CT_T

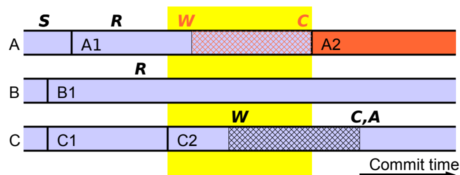


Read-only transactions: Snapshot is always consistent

Timestamp-based Transactions

Update transactions T:

- Updating an object creates a new, most recent version
- On commit, unique commit timestamp CT_T is acquired
- Transaction can commit iff R_T can be extended to $CT_T - 1$
- Validity of newly created object versions starts at CT_T



Read-only transactions: Snapshot is always consistent

Number of extensions required

- Read-only transactions: none (if enough versions are kept)
- Update transactions: ≤ 1 for commit
- In general, at most one extension per read object (caused by concurrent updates)
- Disjoint updates do not increase the number of extensions
- In practice, only a few extensions are required

Are extensions a useful mechanism?

- Simple scenario: read from hotspot at the end of a transaction

Number of extensions required

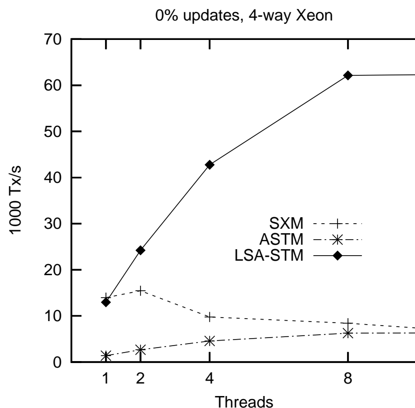
- Read-only transactions: none (if enough versions are kept)
- Update transactions: ≤ 1 for commit
- In general, at most one extension per read object (caused by concurrent updates)
- Disjoint updates do not increase the number of extensions
- In practice, only a few extensions are required

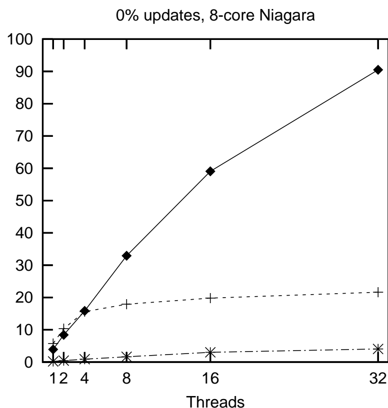
Are extensions a useful mechanism?

- Simple scenario: read from hotspot at the end of a transaction

Performance: Sorted Integer Sets, Linked List

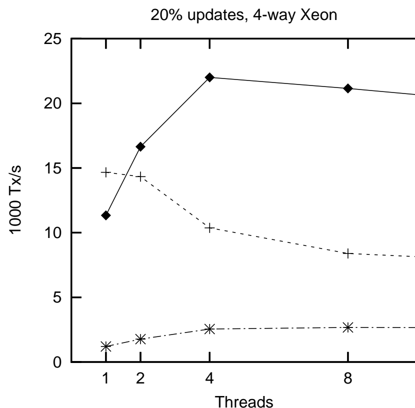
Overall-Throughput(Threads)
Transactions: contains (read-only), insert/remove (update)
250 elements, no early-release

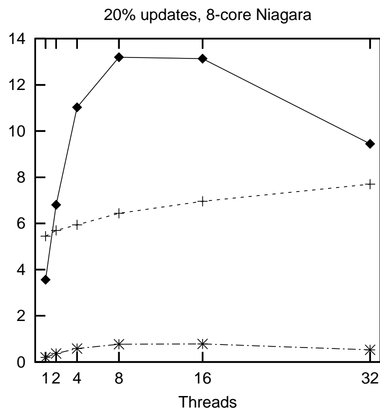




Performance: Sorted Integer Sets, Linked List

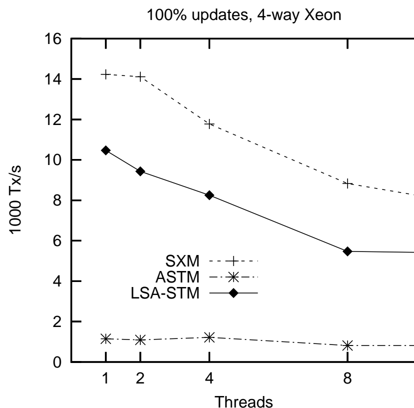
Overall-Throughput(Threads)
Transactions: contains (read-only), insert/remove (update)
250 elements, no early-release

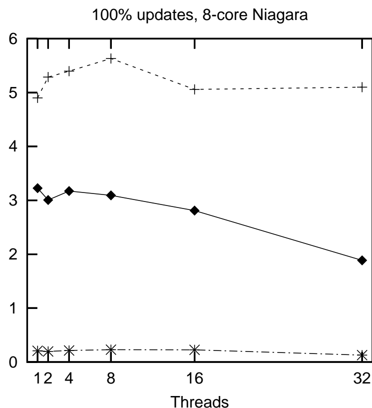




Performance: Sorted Integer Sets, Linked List

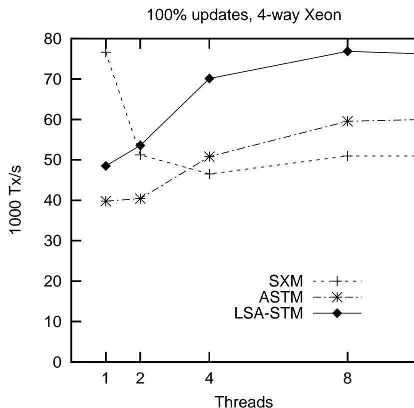
Overall-Throughput(Threads)
Transactions: contains (read-only), insert/remove (update)
250 elements, no early-release

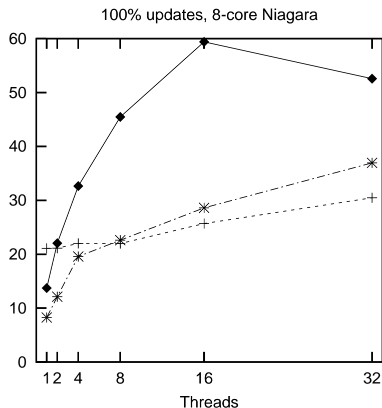




Performance: Sorted Integer Sets, Skip List

Overall-Throughput(Threads)
Transactions: contains (read-only), insert/remove (update)
250 elements, no early-release





Global Commit Time Overhead: Outlook

Current CT implementation: shared integer
$\rightarrow$ can be a problem if update transactions are frequent or on machines with many CPUs

What LSA already does: Avoid accesses to CT

- CT *must* be read at start of transaction
- CT *can* be read on extensions
- Besides that, information from the accessed version is sufficient
- No cache miss for every newly read object

Using fast clocks for CT is possible
Multiple commit times (e.g., per data structure)

Summary

Timestamp-based snapshots / LSA

- Speeds up transactions that read several objects
- Global time base results in some overhead $\longrightarrow$ Current/future work

Extensions

- Can increase level of concurrency
- Optional, on demand

Variety of systems

- Works well on CMT and SMP systems
- Larger systems $\longrightarrow$ Current/future work