Snapshot Isolation for Software Transactional Memory

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

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

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

TRANSACT 2006

Motivation

1. Transactions with large read sets
   → Efficient snapshots to reduce the read overhead
2. Read-only transactions
   → Multiple object versions
3. Workloads with many read/write conflicts
   → Additional support for Snapshot Isolation

Result: SI-STM

Read Overhead

Visible reads:

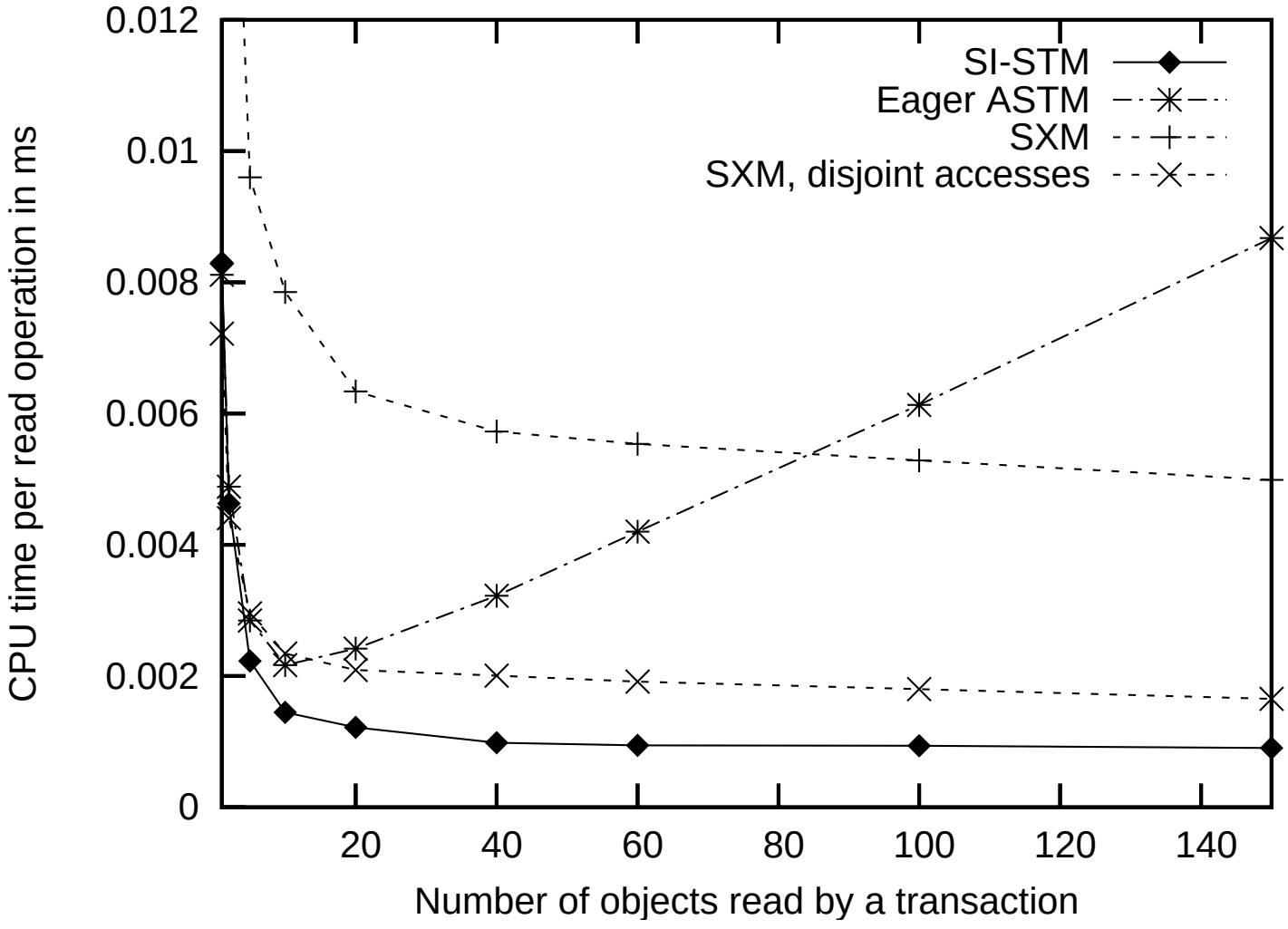
- List of readers $\longrightarrow$ update memory $\longrightarrow$ expensive, contention

Invisible reads:

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

Goal: avoid validation *and* guarantee consistency

Read Overhead (2)



object-based STMs with Java implementations, 4-way Xeon
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 resource allocations?
  - Allocations size/count thresholds
- Before everything that fails if data is inconsistent?
  - Plenty of checks required → lots of validations

Problems: Complex runtime support, libraries

SI-STM: 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

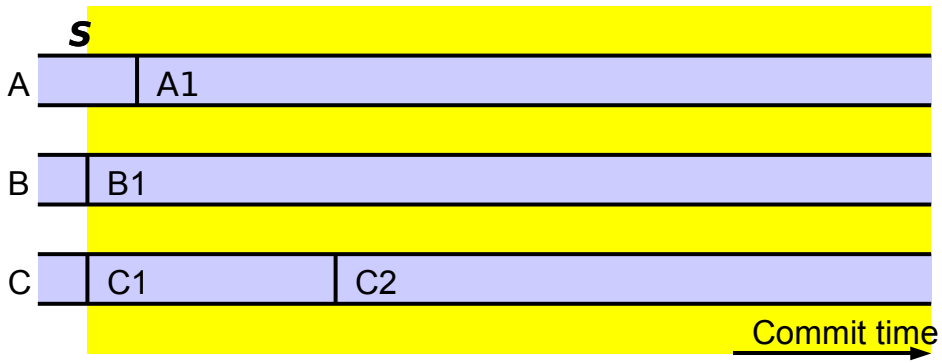
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

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 $\rightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

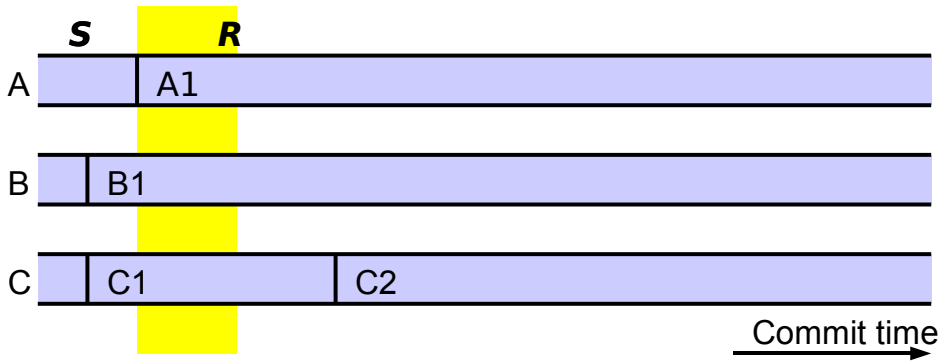
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

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 $\rightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

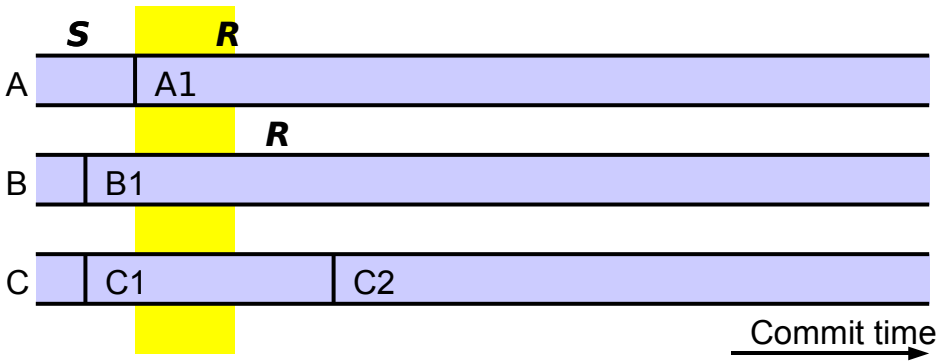
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

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 $\rightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

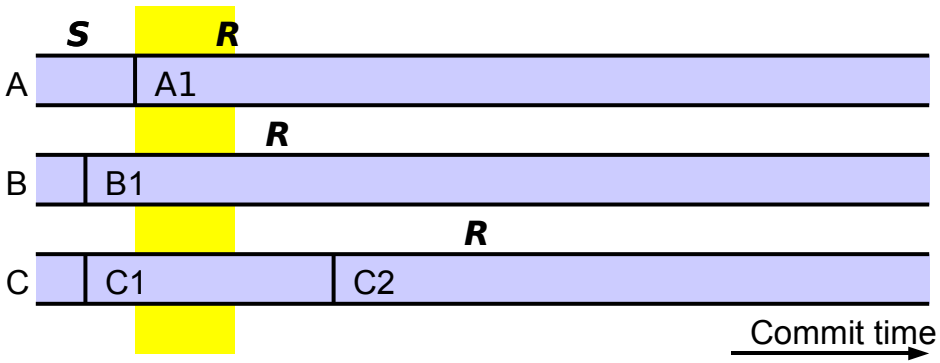
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

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 $\rightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT

STM objects have multiple versions

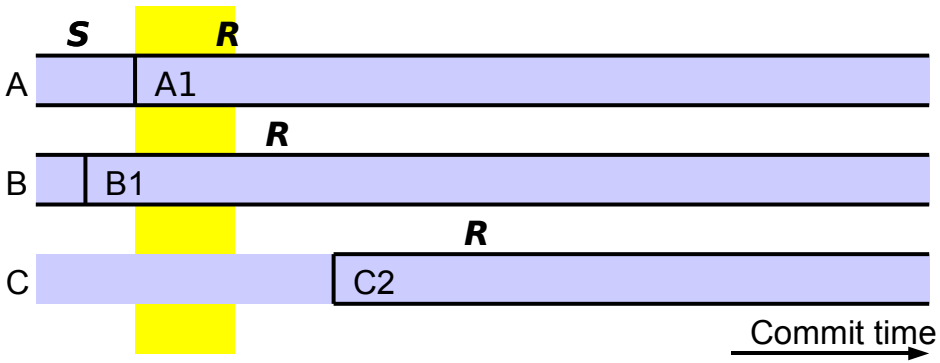
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

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 $\rightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT
STM objects have multiple versions

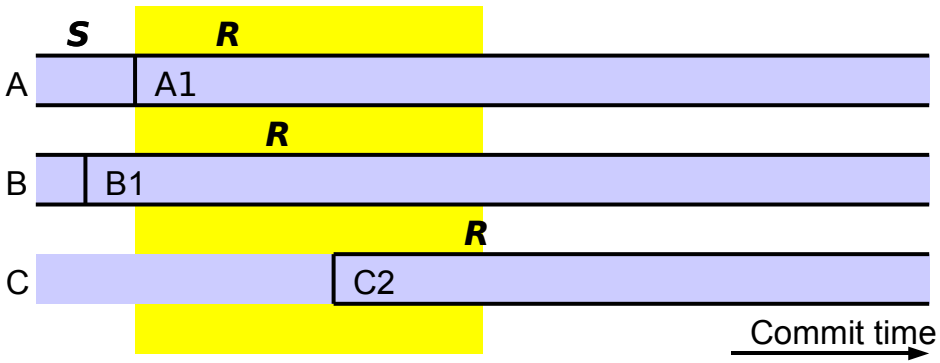
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

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 $\rightarrow$ possibly larger upper bound for R_T



Timestamp-based Snapshots

Time base: global commit time CT
STM objects have multiple versions

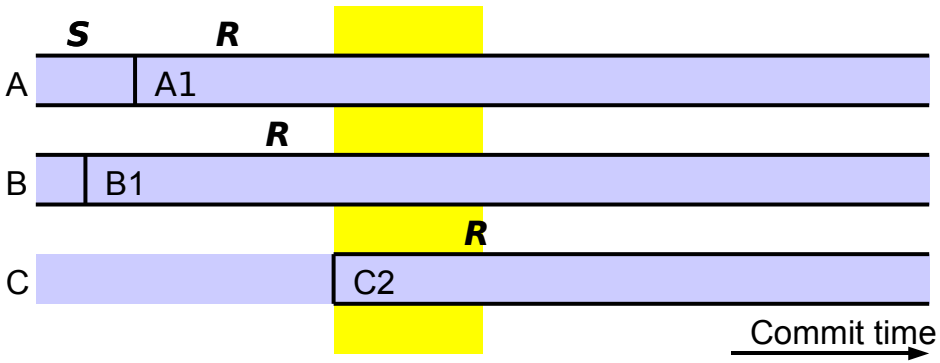
- Each version has a validity range R (w.r.t. CT)
- If most recent version, upper bound undefined

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:

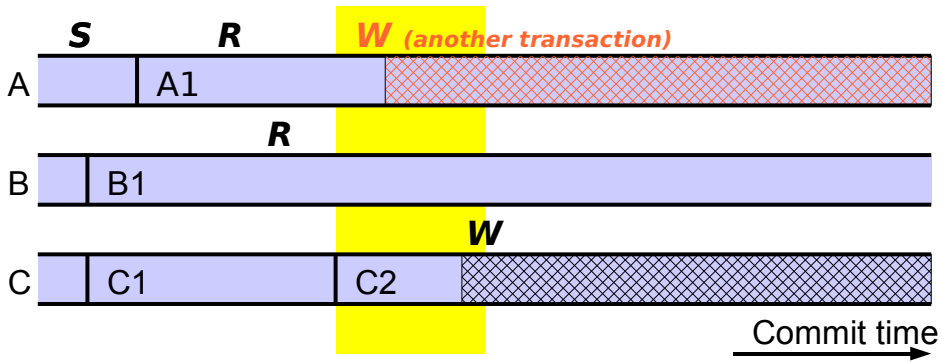
- Validity ranges are recomputed $\rightarrow$ possibly larger upper bound for R_T



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



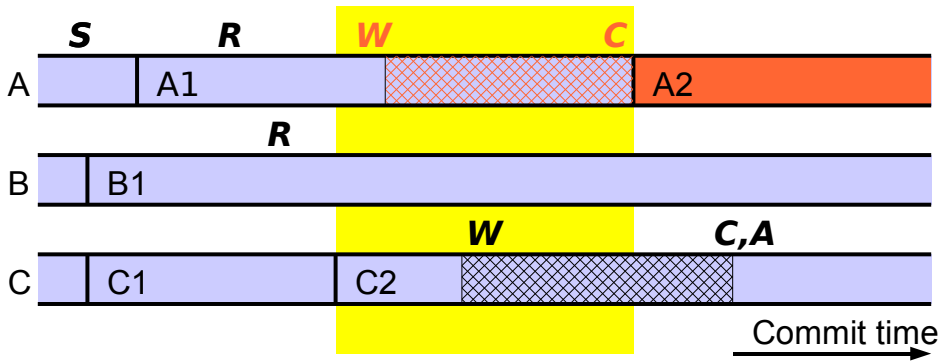
DSTM-style locators:

- Visible writes, only one active update transaction per object
- 1-2 consistent versions plus optional array with older versions

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



DSTM-style locators:

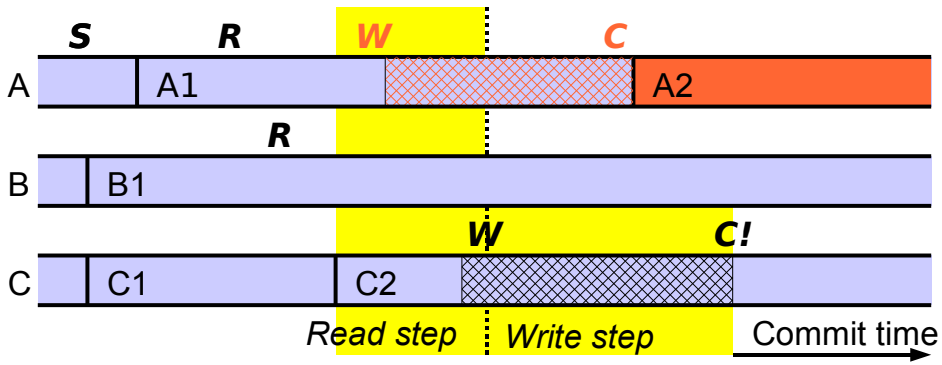
- Visible writes, only one active update transaction per object
- 1-2 consistent versions plus optional array with older versions

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

Snapshot Isolation (SI)

- Isolation level weaker than serializability
- All reads must form a consistent snapshot (read step)
- Updates (write step) must start during the read step
- Only the write step's validity must extend to the commit time



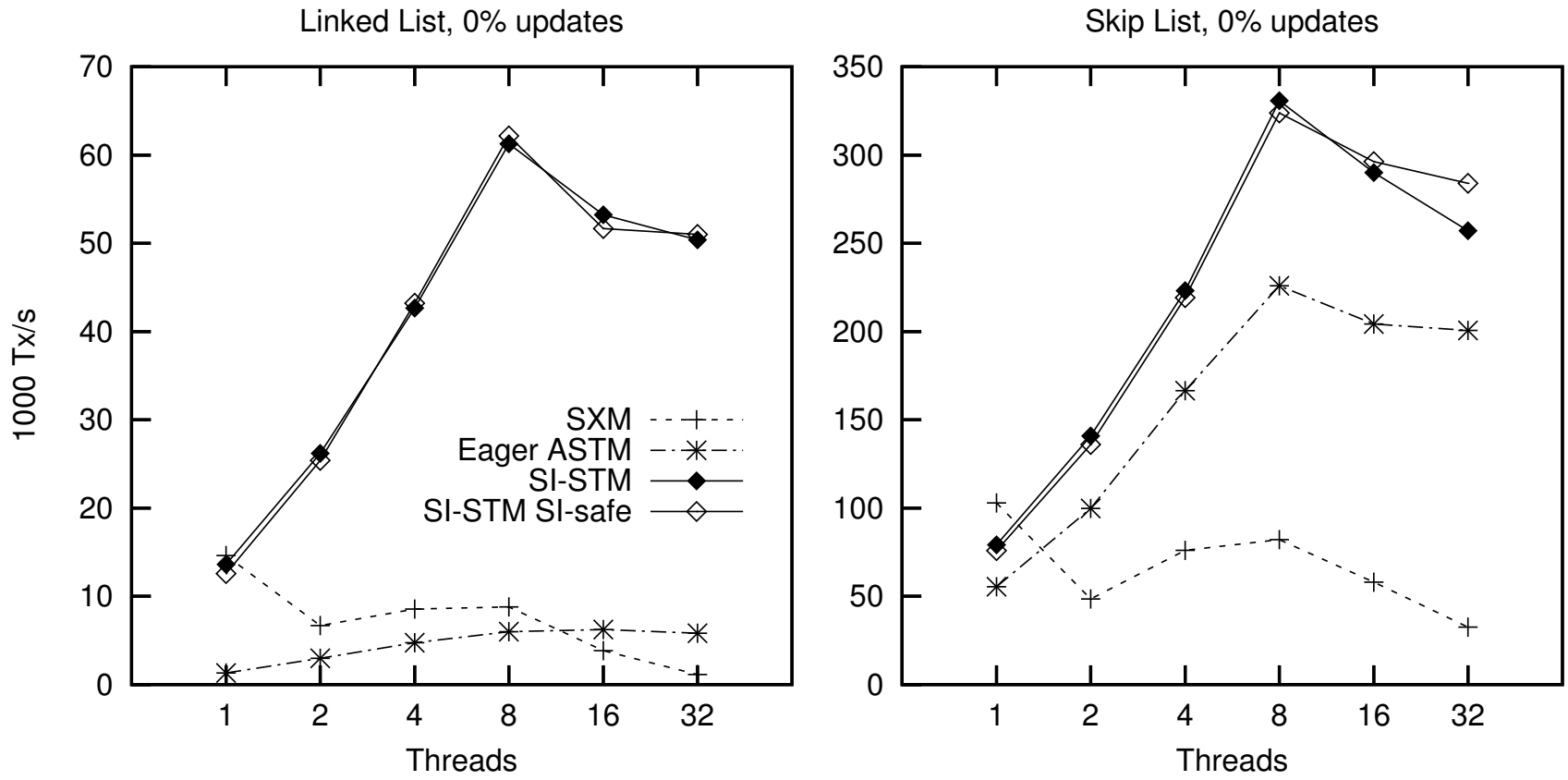
We use Semantic Correctness[1] to make transactions "SI-safe":

- Basic programmer task: check whether other transactions' write steps interfere with the read step's postcondition
- If interference, add dummy writes
  (sorted linked list, add() and remove(): 1 dummy write)

[1] S. Lu, A. Bernstein, and P. Lewis. *Correct execution of transactions at different isolation levels.*

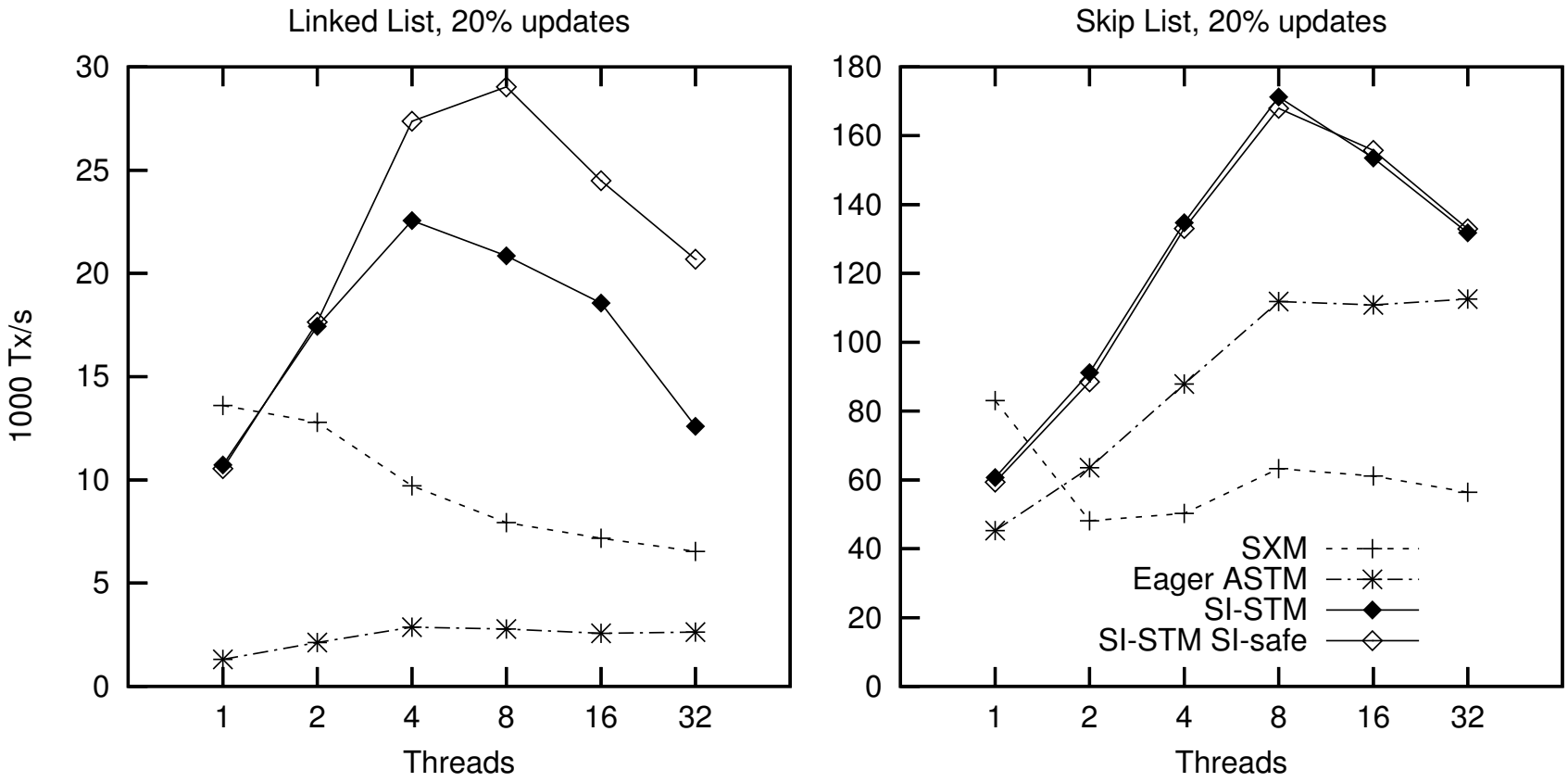
Performance: Sorted Integer Sets

Overall-Throughput(Threads)
Transactions: contains (read-only), insert/remove (update)
250 elements, 4-way Xeon, no early-release



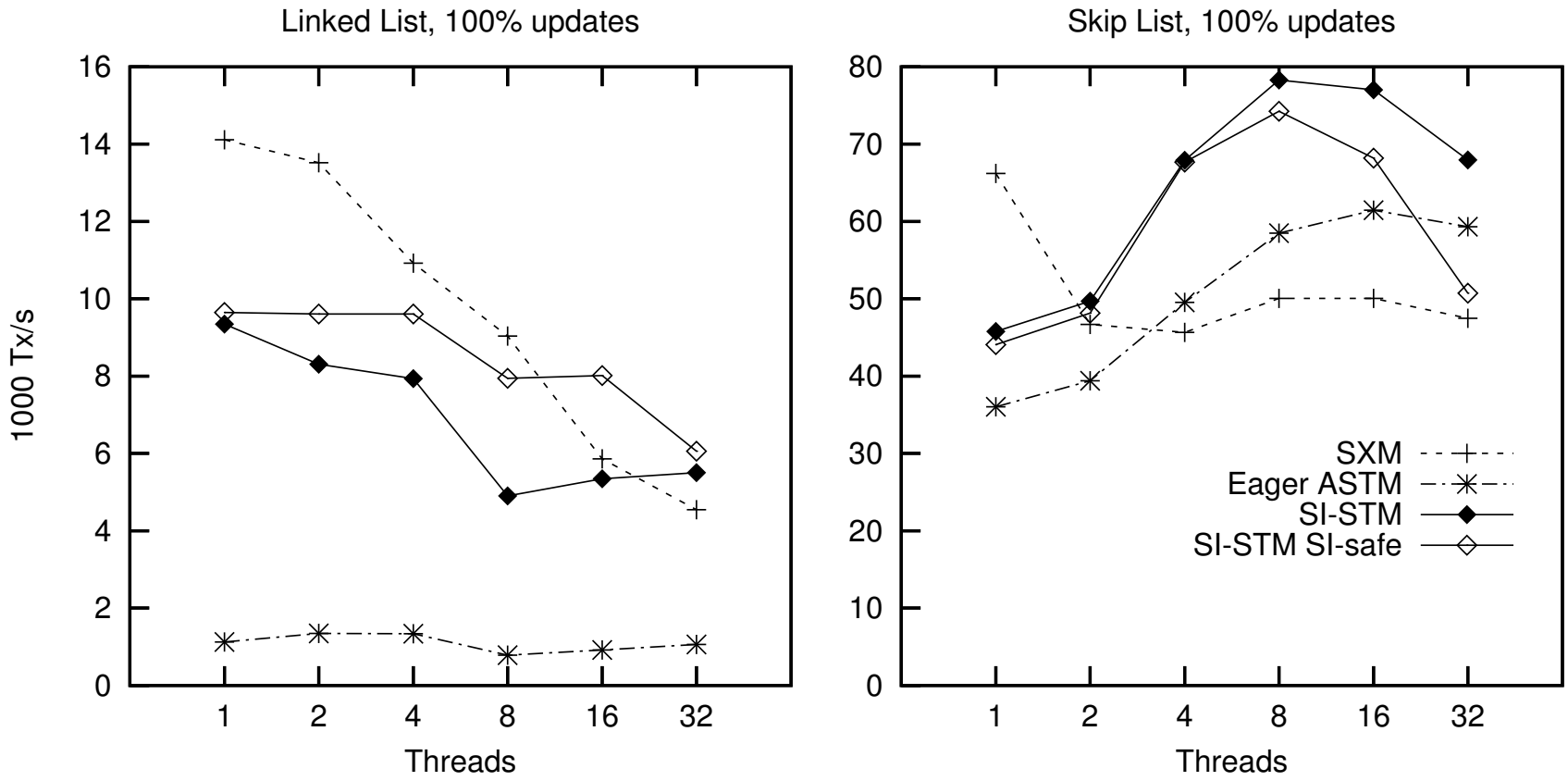
Performance: Sorted Integer Sets

Overall-Throughput(Threads)
Transactions: contains (read-only), insert/remove (update)
250 elements, 4-way Xeon, no early-release



Performance: Sorted Integer Sets

Overall-Throughput(Threads)
Transactions: contains (read-only), insert/remove (update)
250 elements, 4-way Xeon, 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

Using fast clocks for CT is possible

- Does not increase the number of R_T extensions

Multiple commit times (e.g., per data structure)

- Decreases CT contention, partitions data and updates
- CT can be used for quick validation for partitions

Avoid reading CT

- Reads can be performed "in the past" (e.g., time of last read)
- For low conflict workloads with lots of disjoint updates

Summary

Timestamp-based snapshots

- Speed up transactions that read several objects
- Global time base results in some overhead $\longrightarrow$ Future work

Multiple versions

- Speed up read-only (and SI) transactions
- Usefulness of extra versions depends on length of transaction, update frequencies, and when hotspots are accessed

Snapshot isolation

- For careful non-expert programmers
- However, only some transactions/workloads
- Programmer can choose between linearizability and Snapshot Isolation