SMV: Selective Multi-Versioning STM

DMITRI PERELMAN
IDIT KEIDAR

Technion
Israel Institute of Technology

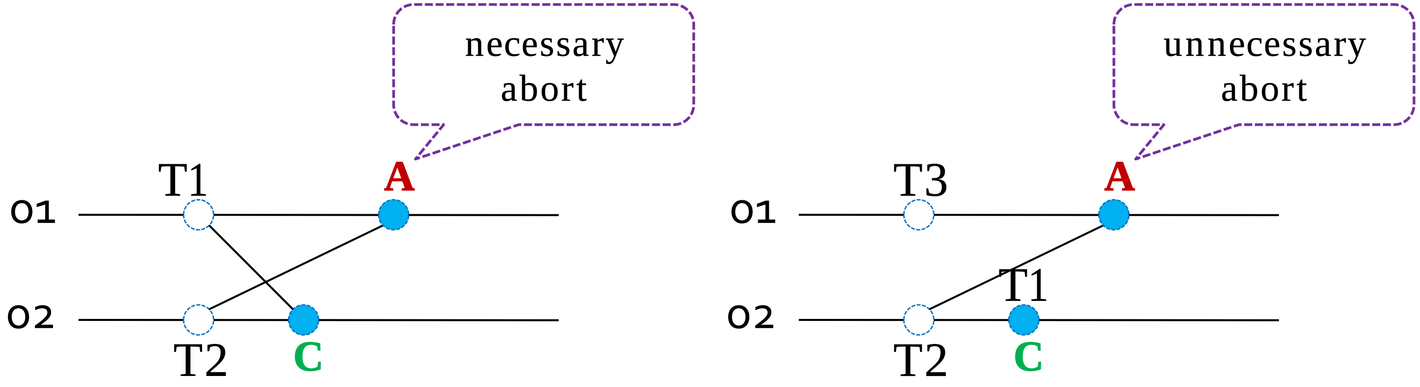
Agenda

- Introduction and problem statement
  - reduce the number of aborts
    - memory consumption
    - invisible reads
- SMV algorithm
  - keeps versions that can help save aborts
  - automatically removes others
- Preliminary evaluation
  - good for read-dominated workloads

Forceful aborts

- Aborting transactions is bad
  - work is lost
  - resources are wasted
  - overall throughput decreases
  - livelock

Unnecessary aborts

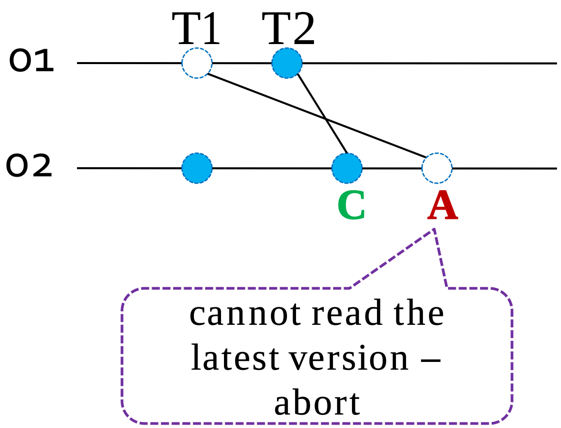


- Sometimes aborts are necessary
  - continuing the run would violate correctness
- And sometimes they are not
  - the suspicion is unjustified

Multi-Versioning in STM

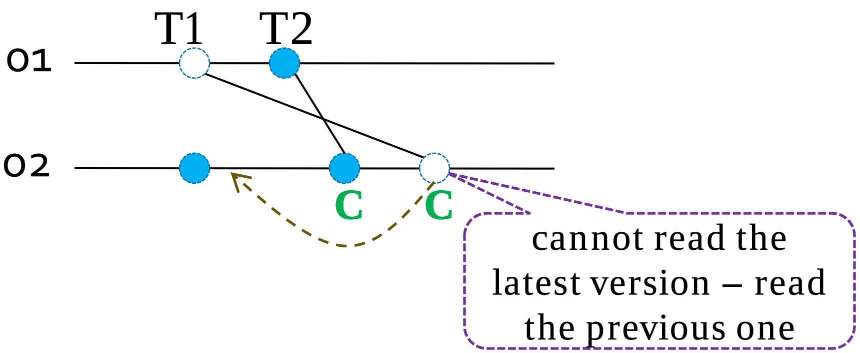
- Keeping multiple versions can prevent aborts

Single-versioned STM



cannot read the latest version – abort

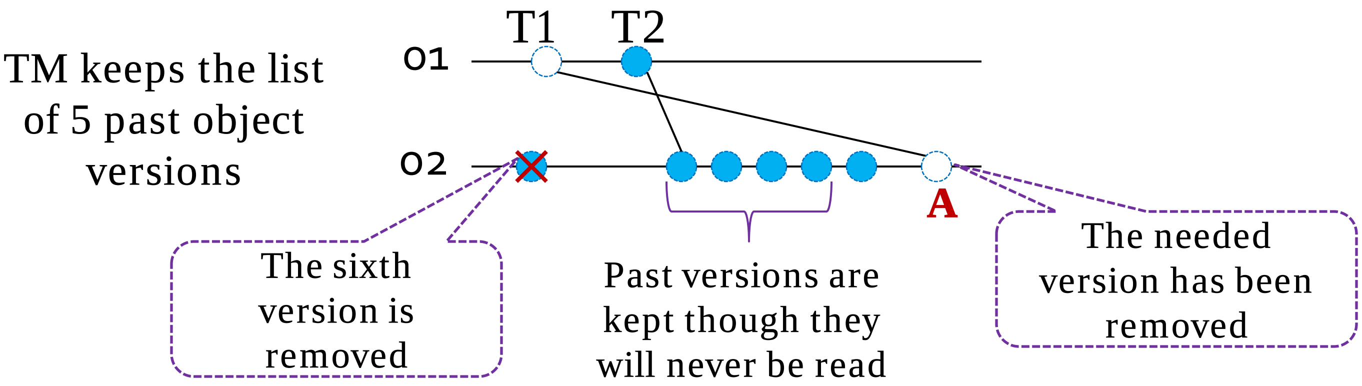
Multi-versioned STM



cannot read the latest version – read the previous one

GC challenge

- Must clean up the old versions
- Many existing TMs keep a list of n past versions
  - some kept versions are useless
  - some potentially useful versions are removed



TM keeps the list of 5 past object versions

The sixth version is removed

Past versions are kept though they will never be read

The needed version has been removed

Visibility challenge

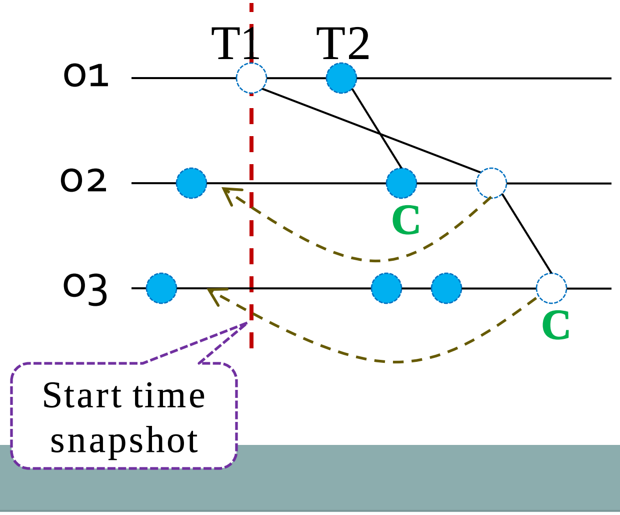
- Changes in memory accessed by other transactions
  - demand the use of costly mechanisms (e.g., volatile variables)
- We want invisible readers
  - do not change data that can be read by others
  - avoid cache thrashing

Agenda

- Introduction and problem statement
  - reduce the number of aborts
    - memory consumption
    - invisible reads
- SMV algorithm
  - keeps versions that can help save aborts
  - automatically removes others
- Preliminary evaluation
  - good for read-dominated workloads

SMV design principles

- A txn is aborted if:
  - update txn: an object from the read-set is overwritten (like most other STMs existing today)
  - read-only txn: (almost) never – commits in a lock-free manner
- T_i reads the latest object value written before T_i starts
- Versions are kept as long as they might be needed
- Read-only transactions are invisible

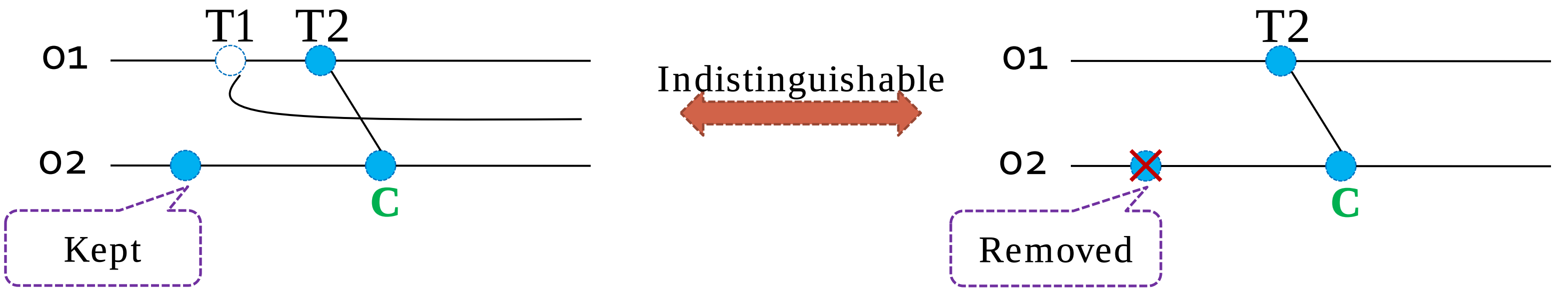


SMV design principles – GC challenge

- A version is removed when it has no potential readers

- Readers are invisible

- No transaction can know whether a given version can be removed
  - explicit GC is not possible



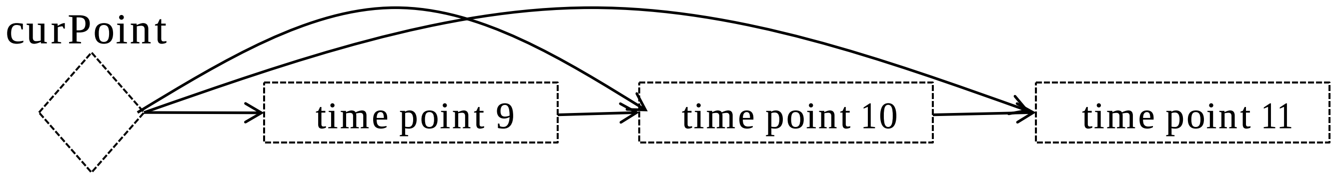
Automated GC in SMV

- Solution: use auxiliary GC threads provided by managed memory systems
  - remove unreachable object versions

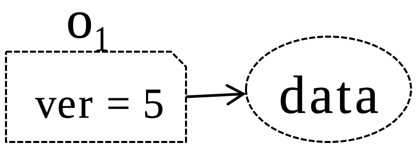
- Read-only transactions are invisible to other transactions, but visible to the “see-all” GC threads
  - theoretically visible
  - practically invisible (GC threads run infrequently)

SMV time progress

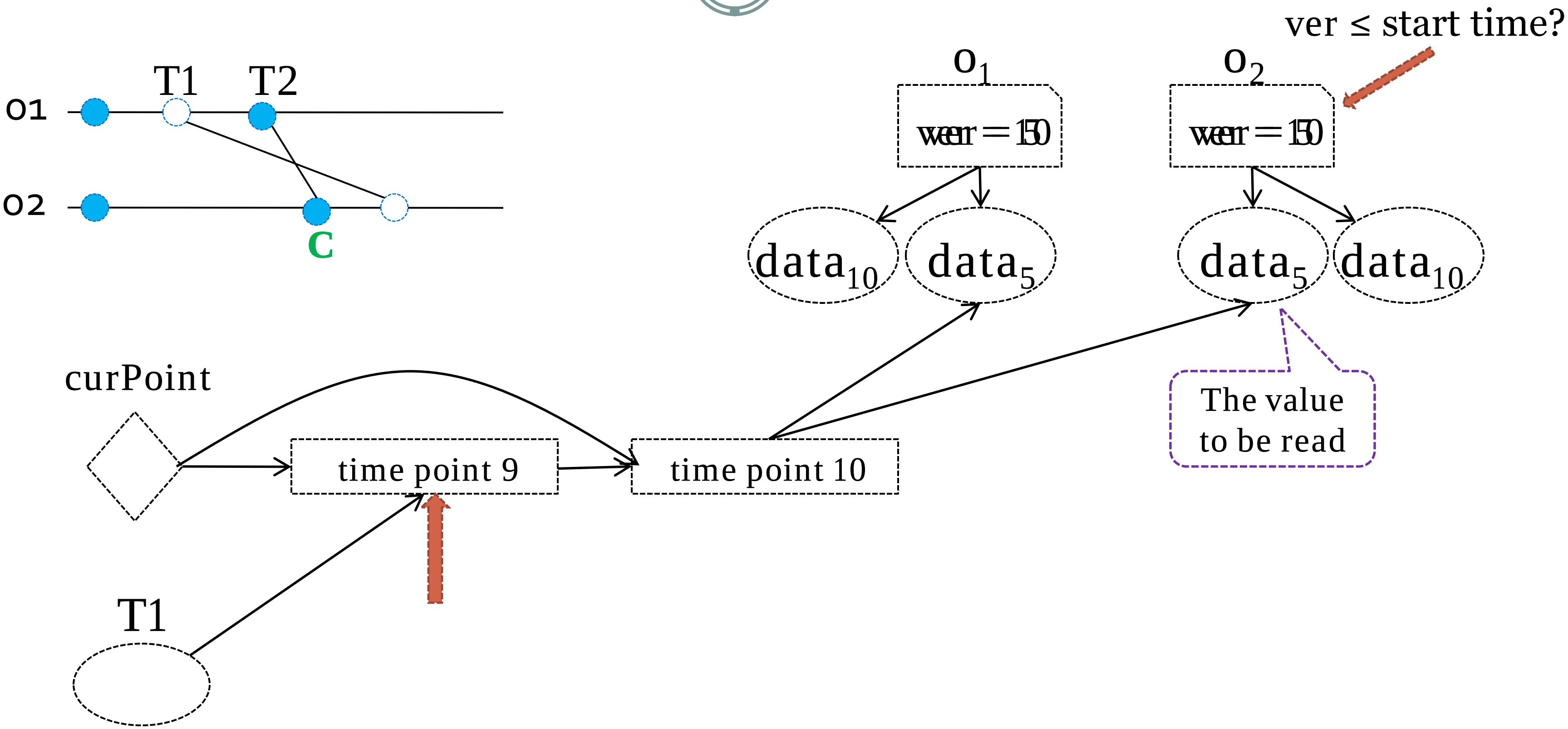
- Logical version clock
  - incremented by update transactions upon commit
  - implemented as a linked list of time points



- Object handles
  - hold versioned locks ala TL2
  - point to the latest object version only



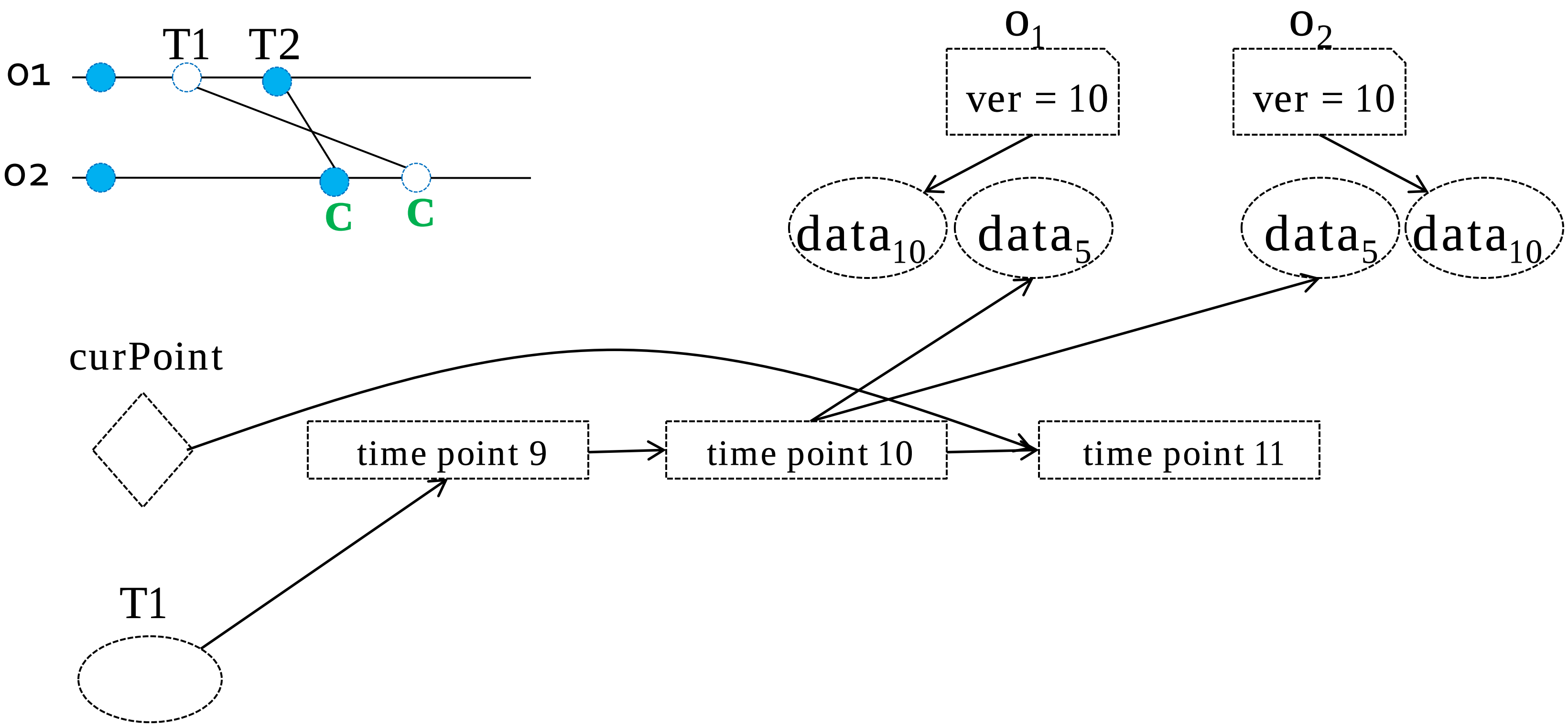
Selective Multi-Versioning STM – overview



The value to be read

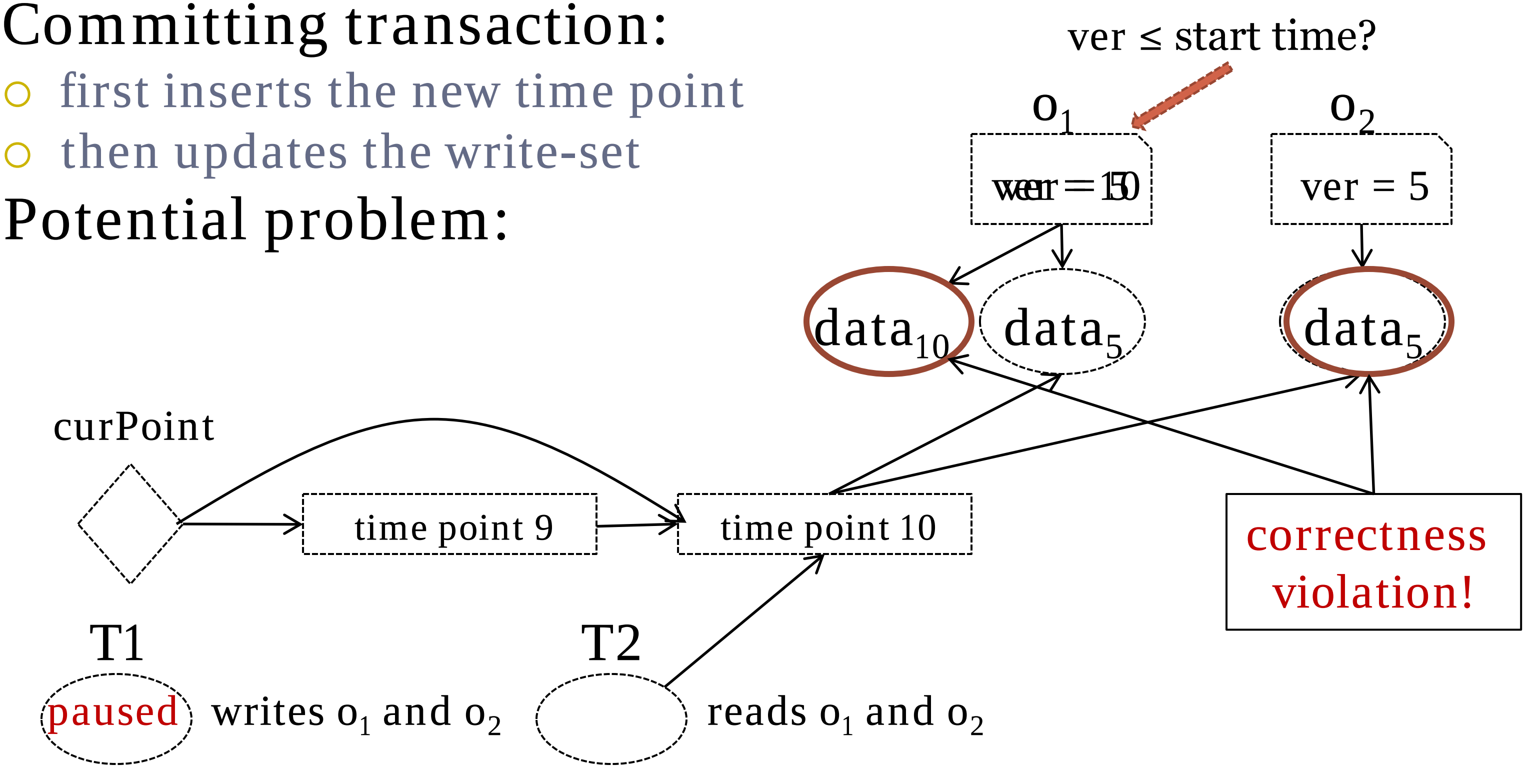
Selective Multi-Versioning STM – GC overview

- Old versions are kept as long as they have potential readers
  - after that they are garbage collected automatically



“Unready” time points issue

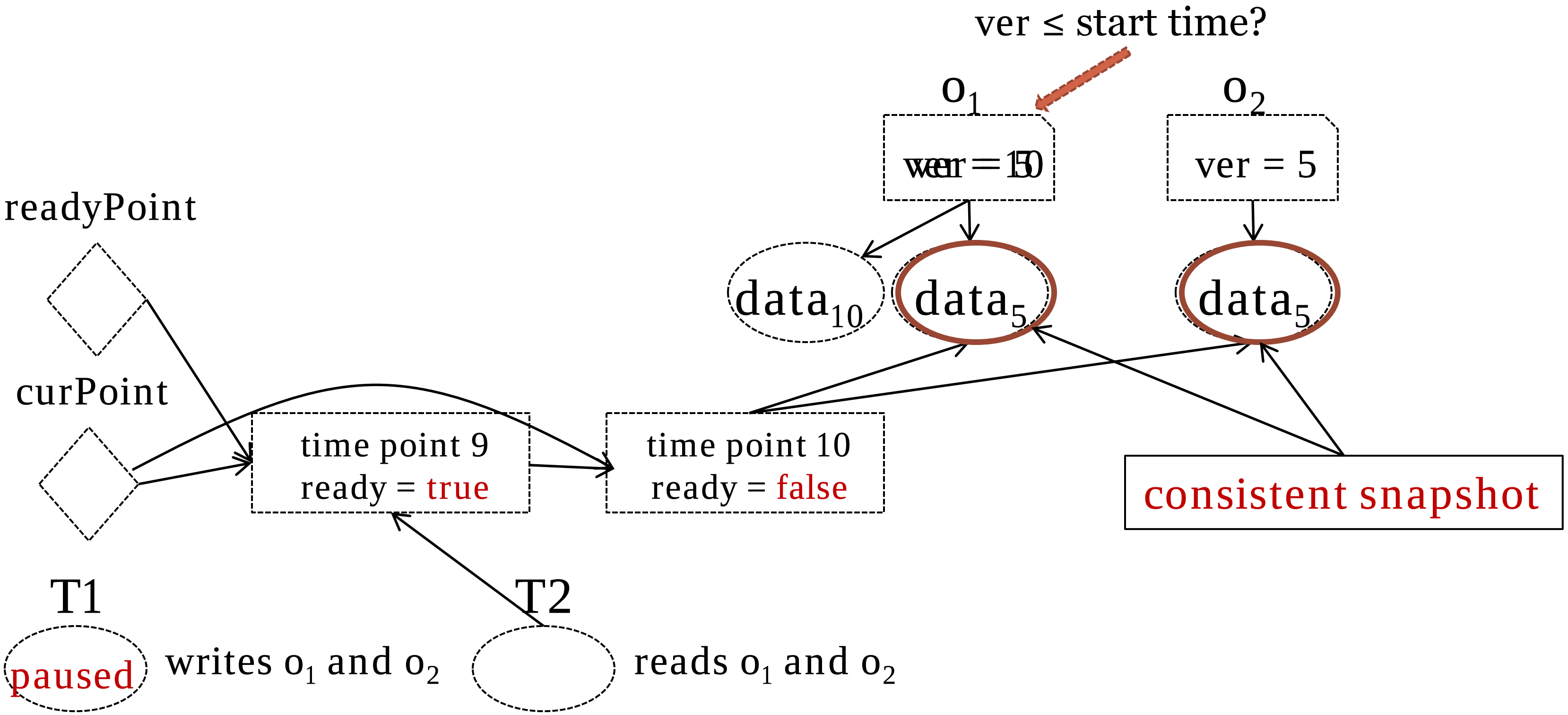
- Committing transaction:
  - first inserts the new time point
  - then updates the write-set
- Potential problem:



- A similar problem is the reason for using locks + double checking in TL2 (each read is pre- and post-validated)

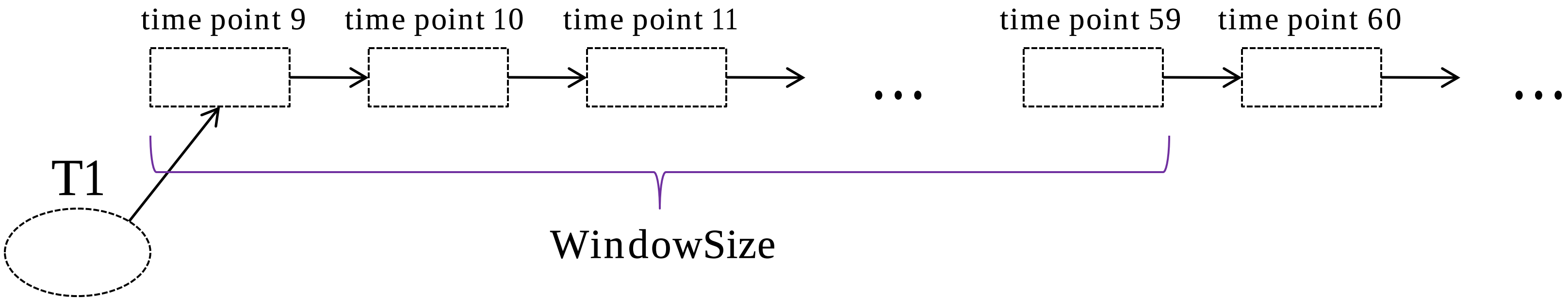
“Unready” time points solution

- Each time point has a boolean *ready* flag
  - true when all the objects are updated
- *readyPoint* points to the latest time point in the ready prefix



Limiting time point traversals

- The number of traversed time points might be large
  - a long read-only txn interleaves with a lot of short update txns
- Limit this number
  - the txn is aborted after traversing WindowSize time points
- Breaks the guarantee of unabortable read-only txns
  - but improves performance



Agenda

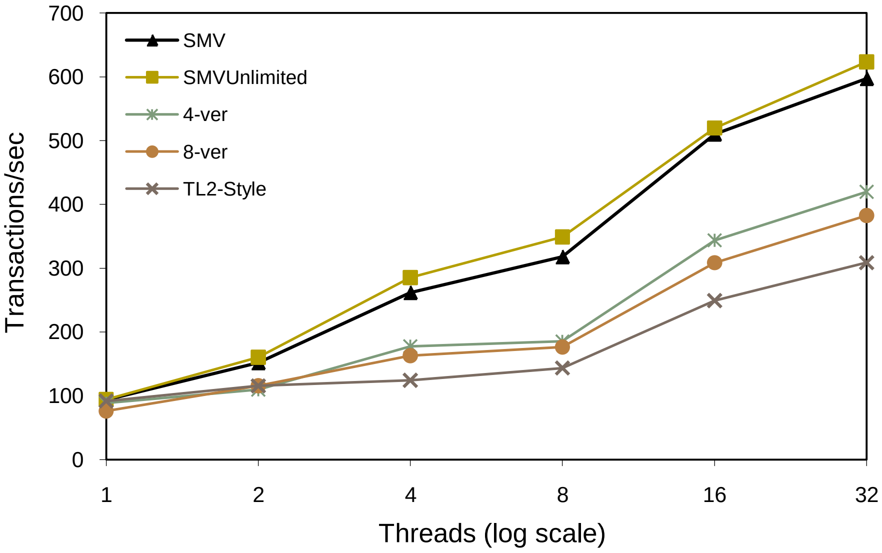
- Introduction and problem statement
  - reduce the number of aborts
    - memory consumption
    - invisible reads
- SMV algorithm
  - keeps versions that can help save aborts
  - automatically removes others
- Preliminary evaluation
  - good for read-dominated workloads

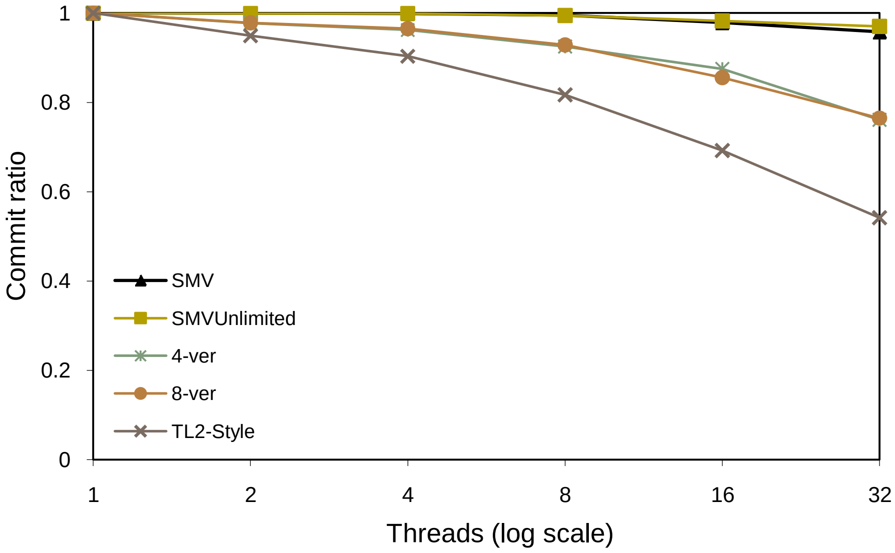
Preliminary evaluation – experiment setup

- STMBench7 evaluation framework – Java version
  - read-dominated and read-write workloads support
- Implemented the following algorithms:
  - SMV (WindowSize = 100)
  - SMVUnlimited (WindowSize = ∞)
  - TL2-style – mimics the basic behavior of TL2
  - k-versioned – each object keeps *k* versions (like in LSA)
- Did not use the software optimizations used in the original algorithms
  - common code platform for comparing the algorithmic issues only

Read-dominated workloads

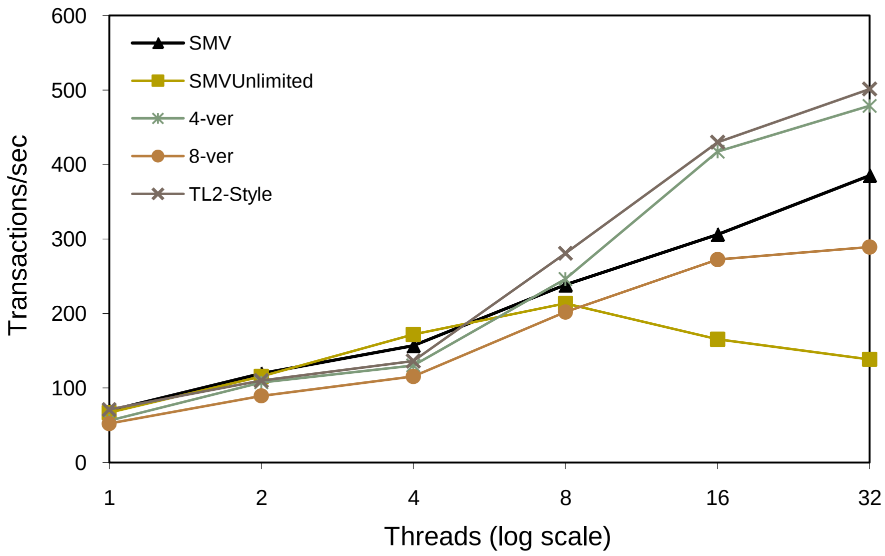
- Emphasize the strong sides of SMV:
  - intensive use of old object versions by read-only txns
  - read-only txns do not need to traverse many time points

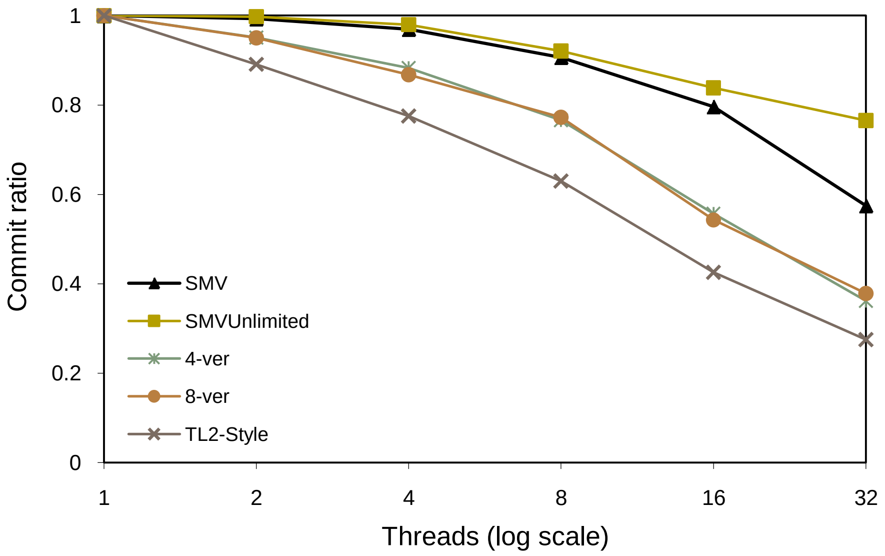




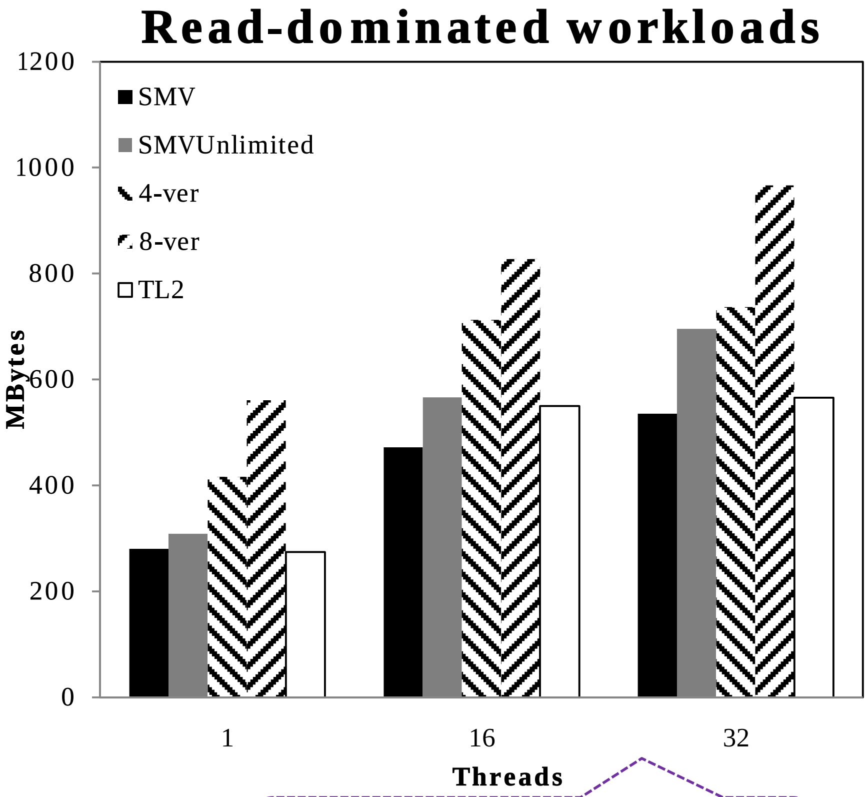
Read-write workloads

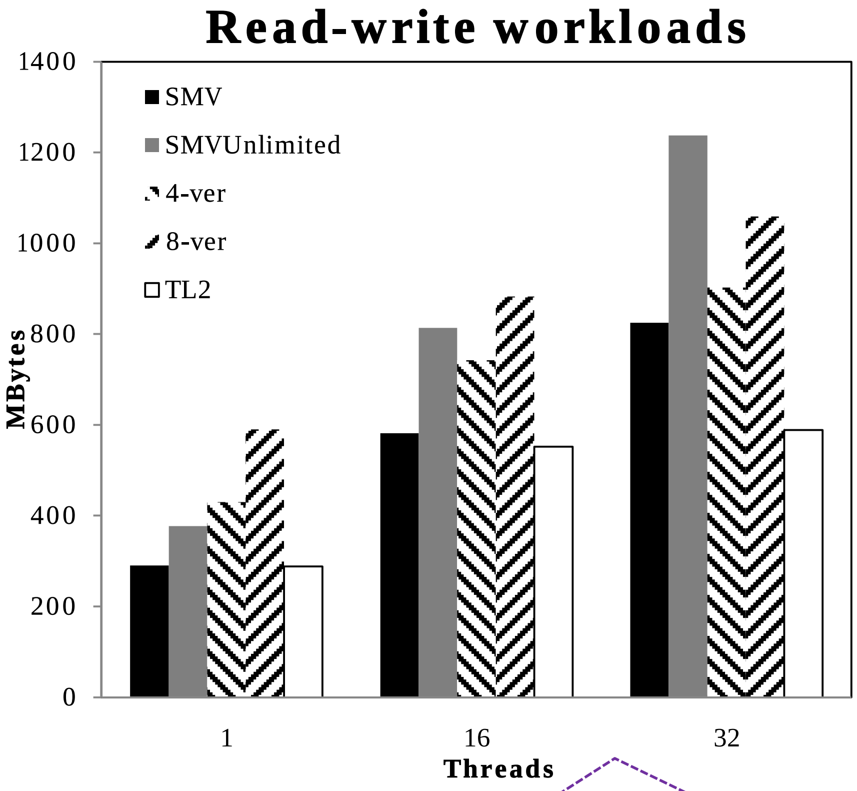
- Present the worst-case scenario for SMV:
  - update txns cannot leverage multiple versions (low commit-ratio)
  - read-only txns traverse long time point list suffixes (high overhead)





Memory consumption





SMV memory consumption is low – for most of the objects keeps last version only

SMVUnlimited memory consumption is high because of long read-only txns

Further work

- Deuce framework
  - field-based synchronization
  - STAMP + STMBench7 benchmarks built-in
- Profiling
  - overhead vs. aborts rate
- GC threads in the non-managed environment
  - fine-tuned GC for txn objects

Thank you