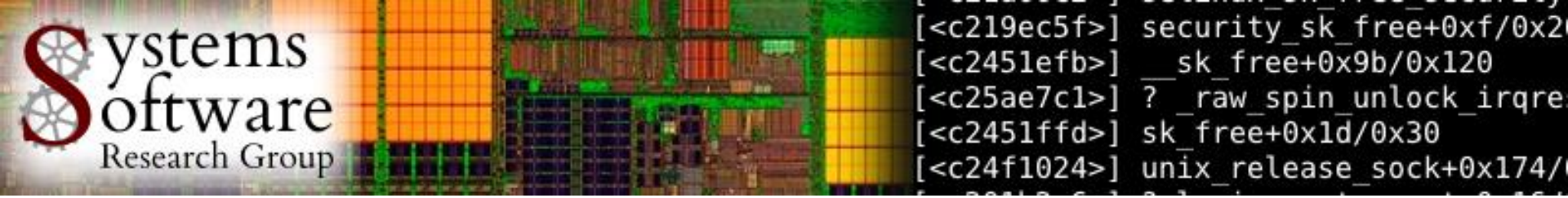


Software Transactional Memory (6)

Mohamed Mohamedin

Chapter 4 of TM Book

Nonblocking STM Systems

- Challenge: Create an STM system without using locks
  - Committing a series of writes atomically without using multi-CAS or locks
  - Making sure a transaction's update are all seen or nothing
  - Conflicts are detected and resolved

DSTM

- First dynamic STM
  - No prior knowledge of the working-set
  - Works on any JVM without modification
  - Explicit Transactions
    - Transactional Objects: TMObject

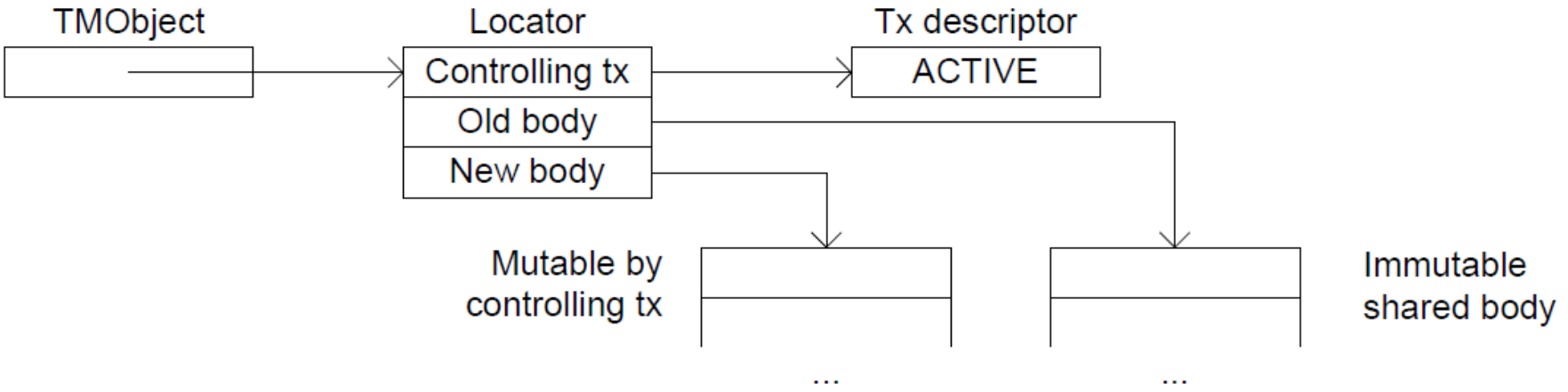
```
class TMObject {
   // Construct new TM object, wrapping obj
   TMObject(Object obj);
   // Request read/write access to TM object,
   // returning current payload
   Object OpenForReadDSTM();
   Object OpenForWriteDSTM();
}
```

DSTM

- How it works
  - Two levels of indirection
  - Applications use TMObjects to refer each object in the program
    - No direct reference is allowed to actual data
    - TMObjects use locators to find the object logical value
    - Locators are immutable
    - TMObject has a reference to a transaction' descriptor

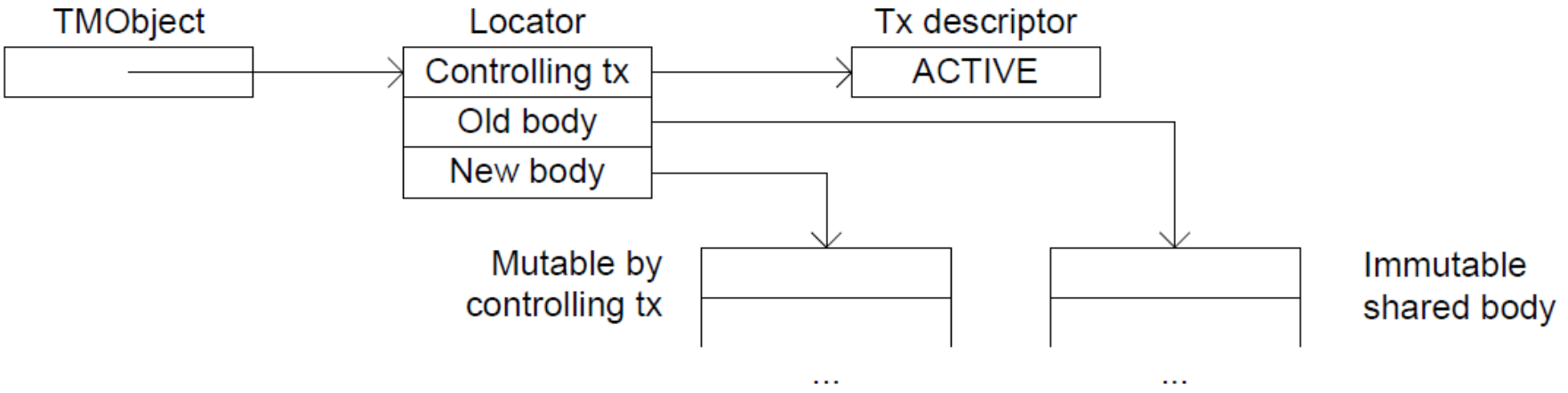
DSTM

- TMObject
  - Point to a locator
  - Locator point to
    - Last transaction opened the object for writing
    - Old value
    - New value



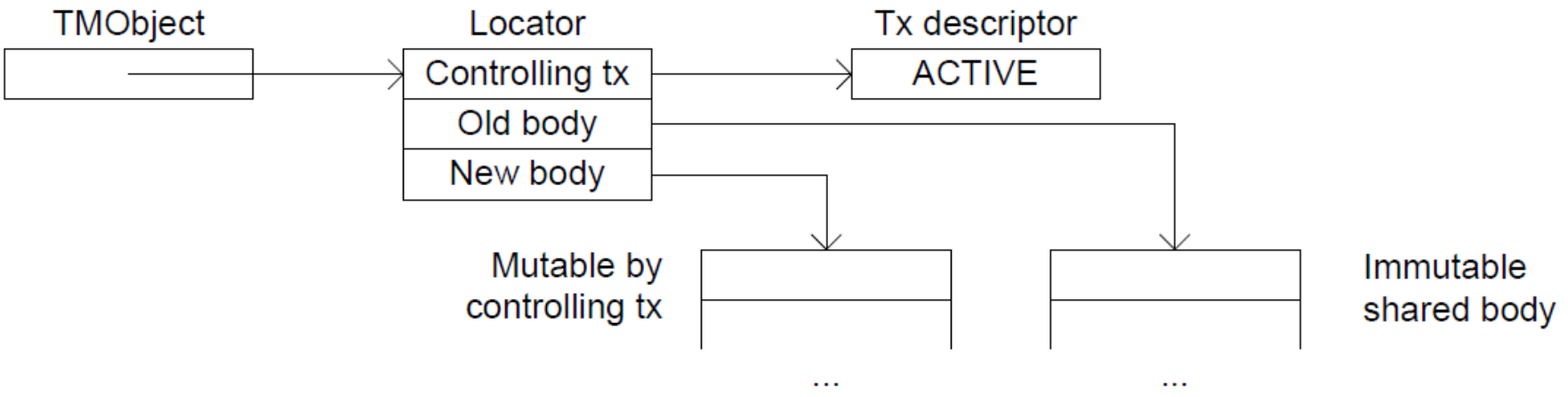
DSTM

- TMObject
  - Locator is immutable
    - To change any of its fields
      - Create a new locator with the new values
      - Using CAS to change TMObject locator's pointer
    - No lock is needed to change the locator



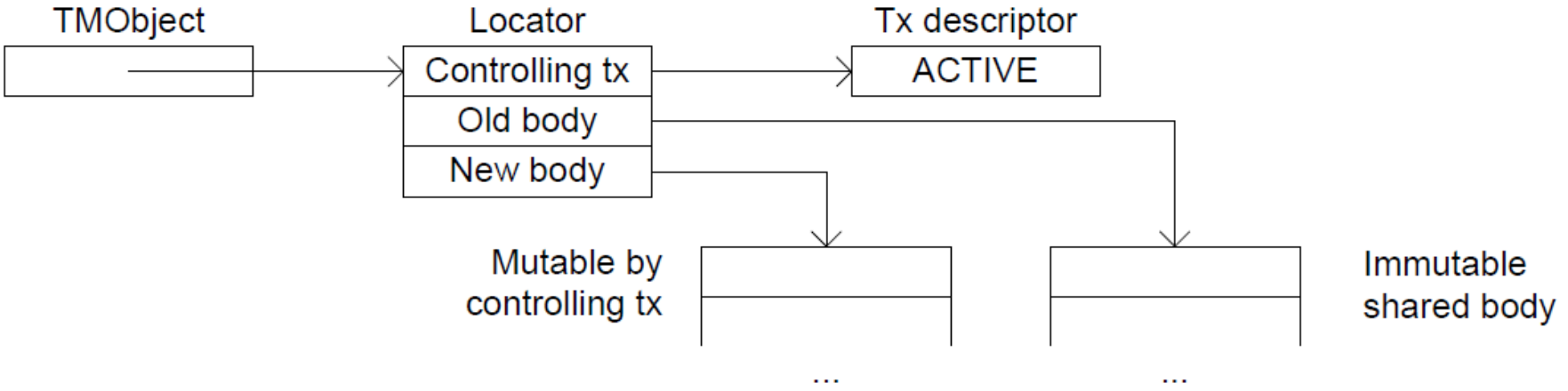
DSTM

- TMObject Logical state
  - Using the Locator & Tx Descriptor status (ACTIVE, ABORTED, COMMITTED)
    - COMMITTED: Follow the new value pointer
    - ACTIVE or ABORTED: Follow the old value pointer



DSTM

- How a transaction is committed?
  - A single CAS is needed to change the Tx Descriptor status
    - All associated object will point to the new value.
- Aborting a transaction is the same
  - Change the status to ABORTED using one CAS

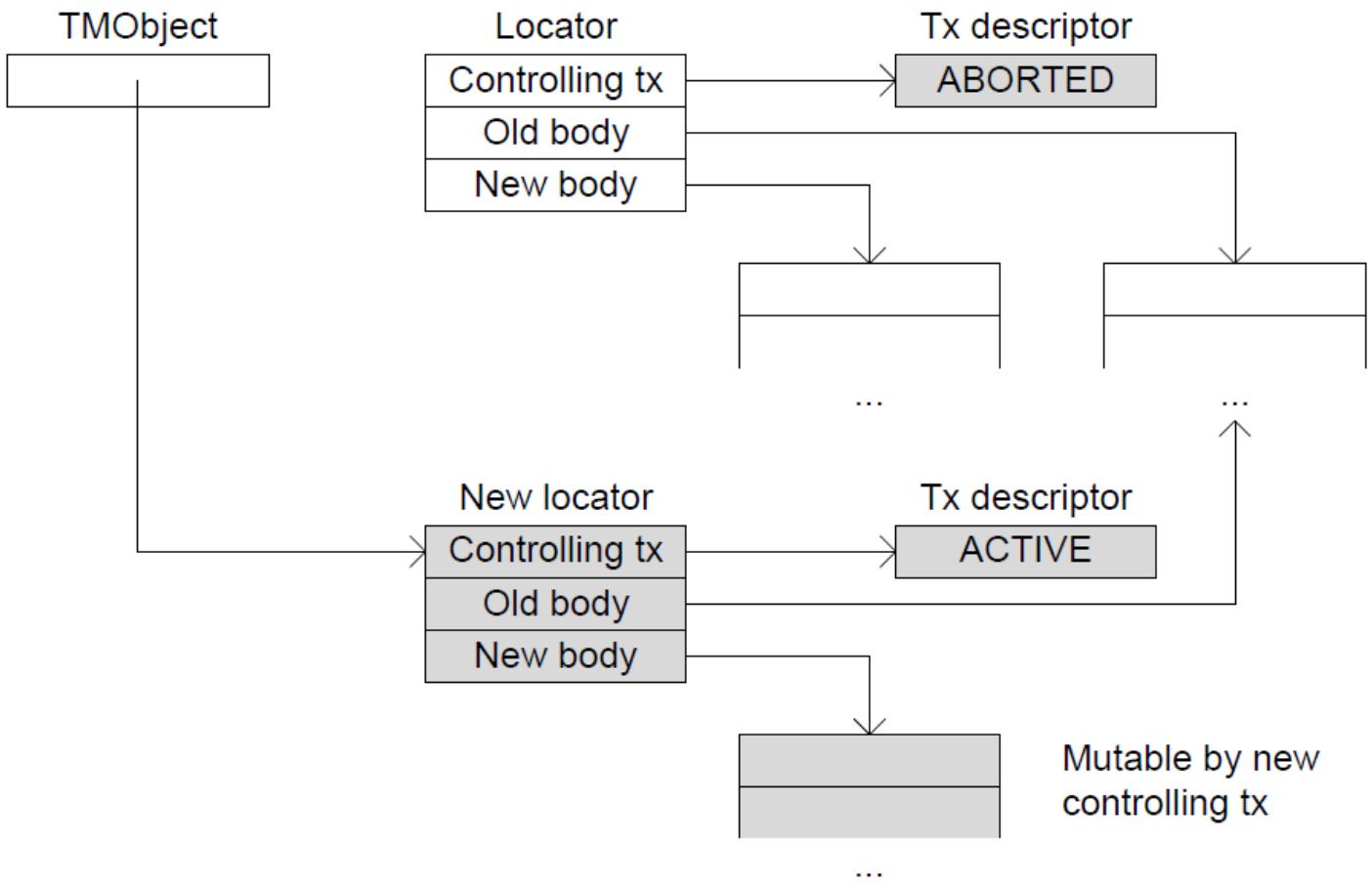


DSTM

- OpenForWriteDSTM
  - Check if the Tx already has the object open for write → then return the new value
  - Else: Take control of the object
    - Abort the current owner transaction if ACTIVE (CAS its status to ABORTED)
    - After gaining control of the object:
      - Create a fresh locator
      - Copy the object's logical value (using the current locator) to the new locator's old value
      - The written value becomes the new locator's new value
      - Use a CAS to change replace the old locator with the new one
      - Note: the logical value of the object remains the same

DSTM

- OpenForWriteDSTM

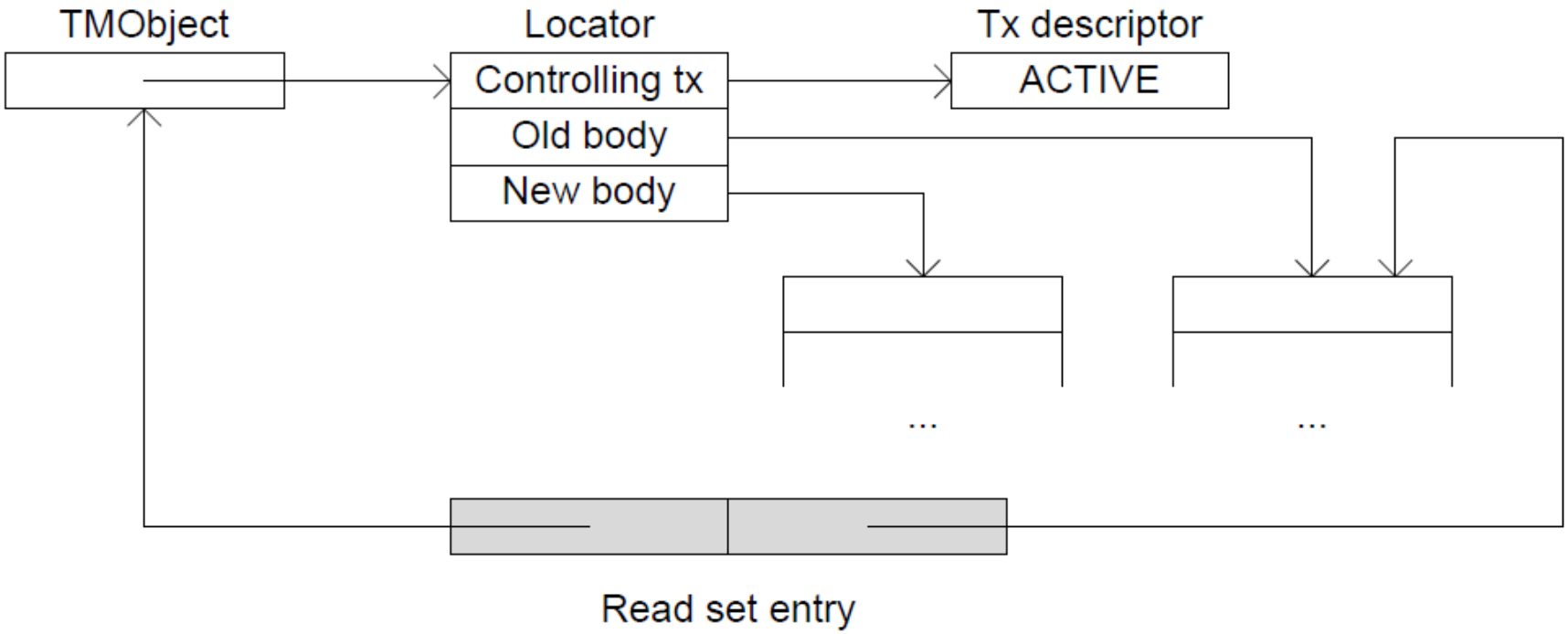


DSTM

- OpenForReadDSTM
  - Determine the object's logical value
  - Add it to the read-set
  - Validate the entire read-set

DSTM

- OpenForReadDSTM



DSTM

- CommitTx
  - Validate the read-set
  - If validation is successful
    - Using one CAS operation, change the Tx status from ACTIVE to COMMITTED

DSTM

- Main Ideas
  - Each object pointing at the transactions descriptor
    - One CAS operation to make changes visible/abort
  - Logical content of an object based on its metadata values
  - Using immutable shared data instead of locking

Non-Blocking STM Designs Taxonomy

- *"When are objects acquired for writing?"*
  - *Eager Acquire*
    - Acquire the object before writing to it
    - Earlier conflict detection (writers are visible)
  - *Lazy Acquire*
    - Only acquire the object during commit procedure for a short time
    - Can avoid livelock
- DSTM?

Non-Blocking STM Designs Taxonomy

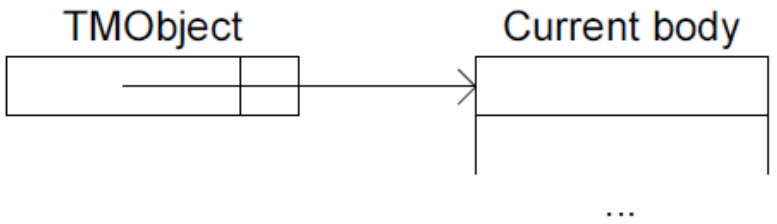
- *"Are readers visible to writers?"*
  - *Invisible readers*
    - Writers are not aware of existing readers
    - No contention between readers
    - Easier to implement
  - *Semi-visible readers*
    - Writers are aware of existing readers BUT cannot identify them
    - E.g., using bitmap-based visible readers
  - *Visible readers*
    - Writers know exactly which transaction is reading
    - Earlier contention-management policies
    - No incremental validation is needed

Non-Blocking STM Designs Taxonomy

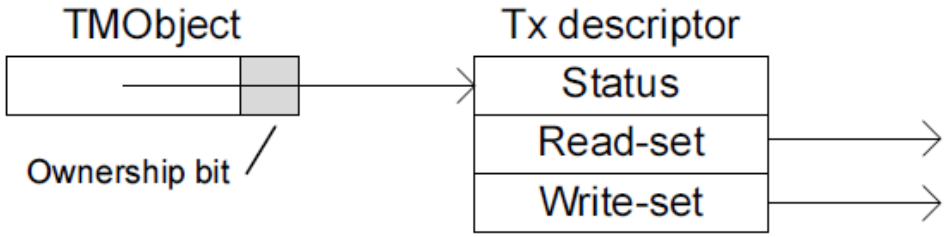
- *"How are objects acquired by transactions?"*
  - TMObject refers to another metadata object
    - Similar to DSTM
  - TMObject refers to a Tx descriptor → current content, OR actual value
    - It has a bit to determine whether it is referencing the actual value or a Tx descriptor
    - Reduce level of indirection from 2 to 1
    - Commit is slower
      - Visits each written object twice: (1) Install the descriptor (2) Remove the descriptor

Non-Blocking STM Designs Taxonomy

- *"How are objects acquired by transactions?"*
  - TMObject refers to a Tx descriptor → current content or actual value



(a) Quiescent



(b) Acquired

Revising STM algorithms we studied so far

REVISION

Transactional Locking

- TL Global Metadata
  - Versioned Locks
    - Per object
    - Lock Table
- TL Private Metadata
  - Read-set: Keeps tracks of locations read
  - Write-set: Keeps tracks of locations written

Transactional Locking

- What is a Versioned Lock
  - Lock + Version
  - Lock gives exclusive access
  - Version is used to detect conflicts
  - Version must be incremented with every new value written to the associated location

Transactional Locking

- STM functions
  - tx_begin(): initialize private metadata
    - Reset read-set and write-set

Transactional Locking

- STM functions
  - tx_write(addr, val):
    - Add the location and new value to the write-set

Transactional Locking

- STM functions
  - tx_read(addr):
    - Find the location in the write-set
      - If found, return the value in the write-set
    - If addr is locked
      - tx_abort()
    - Else
      - add the location and version to the read-set
      - Return the value from the memory

Transactional Locking

- STM functions
  - tx_commit():
    - For each entry in the write-set
      - Acquire the lock associated with that entry
      - If lock is acquired by another thread → tx_abort()
    - //Validate the read-set
    - For each entry in the read-set
      - If (entry.version != version(entry.addr)) → tx_abort()
    - //Now transaction is valid and can be committed
    - For each entry in the write-set
      - *entry.addr = entry.value //write back
      - version(entry.addr) = version(entry.addr) + 1
      - unlock(entry.addr)

Transactional Locking

- STM functions
  - tx_abort():
    - For any locks we acquired so far
      - unlock(entry.addr)

TL with Eager Versioning

- Uses *direct updates, undo-log, and encounter-time locking*
- Acquire a write-lock before writing a location
- Writes directly to memory
  - Keep old replaced value in the undo-log
- If a transaction is aborted, restore memory using the undo log

TL with Eager Versioning

- STM functions
  - tx_begin(): initialize private metadata
    - Reset read-set, and write-set (undo-log)

TL with Eager Versioning

- STM functions
  - tx_write(addr, val):
    - Acquire the location's lock
      - Wait with timeout → abort if timeout
    - Add the location and **current** value to the write-set
    - //Write to the location directly
    - *addr = val

TL with Eager Versioning

- STM functions
  - tx_read(addr):
    - If addr is locked by another transaction
      - Wait with timeout → abort if timeout
    - Else
      - Add the location and version to the read-set
      - Return the value from the memory

TL with Eager Versioning

- STM functions
  - tx_commit():
    - //Validate the read-set
    - For each entry in the read-set
      - If (entry.version != version(addr)) → tx_abort()
    - //Now transaction is valid and can be committed
    - For each entry in the write-set
      - version(addr) = version(addr) + 1
      - unlock(addr)

TL with Eager Versioning

- STM functions
  - tx_abort():
    - For each entry in the write-set
      - //restore memory values
      - *entry.addr = entry.value
      - unlock(entry.addr)

Eager or Lazy Versioning?

- Lazy
  - Locks are acquired for a short period
  - Allows more concurrency
  - Better performance with contention
  - Aborting a transaction is fast
  - BUT:
    - Reads search through the write-set
      - Can be optimized using a hash based write-set or Bloom filters
    - Commit is longer (write-back)

Eager or Lazy Versioning?

- Eager
  - Locks are held for a long period
  - Commit is fast
  - Reads do not search the write-set
  - BUT:
    - Bad performance with contention
    - Restrict concurrency in some scenarios
    - Aborting a transaction is slow

Transactional Locking Issues

- It suffers from *Zombie* transactions
  - Only guarantees external consistency
  - Internal consistency is not guaranteed
    - A transaction can observe an inconsistent state before it is aborted
    - A transaction may not reach validation
      - Infinite loop
      - Unexpected exception
    - Validating the read-set after each read is costly

TL2

Transactional Locking II

- Guarantees internal consistency
  - Using a Global Clock
  - Now versions are related to the Global Clock
  - Versions has a global meaning now!
    - Global Versioning vs. Local Versioning

Transactional Locking II

- Read the Global Clock at the beginning and store it to a local variable (RV)
- At each read, confirm the location version is less than or equal to RV

Transactional Locking II

- Metadata
  - Global
    - Global clock
    - Lock table (versioned locks)
  - Private
    - Read Version (RV) or start-time
    - Read-set: contains addresses being read
    - Write-set: contains addresses being written and their new values

Transactional Locking II

- tx_begin()
  - Reset read-set and write-set (write-buffer)
  - RV = Global-Clock

Transactional Locking II

- tx_write(addr, value)
  - Add (or update) the addr and value to the write-set

Transactional Locking II

- tx_read(addr)
  - Find the addr is in the write-set
    - If found, return the value buffered in the write-set
  - v1 = version(addr)
  - CFENCE
  - val = *addr
  - CFENCE
  - v2 = version(addr)
  - if (v2 <= RV && v1 == v2 && lock(addr) == 0)
    - Add addr to read-set
    - Return val
  - Else
    - tx_abort()

Transactional Locking II

- tx_commit()
  - For each entry in the write-set
    - Acquire the lock associated with that entry
    - If lock is acquired by another thread → tx_abort()
  - WV = Atomic_inc(Global-Clock)
  - //Validate the read-set
  - For each entry in the read-set
    - If (version(entry.addr) > RV || lock(entry.addr) == 1) → tx_abort()
  - //Now transaction is valid and can be committed
  - For each entry in the write-set
    - *entry.addr = entry.value //write back
    - version(entry.addr) = WV
    - lock(entry.addr) = 0 //unlock

Transactional Locking II

- tx_abort()
  - For any locks we acquired so far
    - lock(entry.addr) = 0 //Unlock

Transactional Locking II

- Read-Only Transactions
  - Can be highly optimized
  - No need for a read-set!
  - tx_read(addr)
    - v1 = version(addr)
    - CFENCE
    - val = *addr
    - CFENCE
    - v2 = version(addr)
    - if (v2 <= RV && v1 == v2 && lock(addr) == 0)
      - Return val
    - Else
      - tx_abort()

Timestamp Extension

- If an inconsistency found in a tx_read
  - Read the Global Clock again (RV`)
  - Revalidate the read-set using the original RV
  - If validation is successful → RV = RV`
    - No abort is needed!

Issues with TL2

- Global Clock
  - Introduce a bottleneck
    - High contention since all writing transactions are incrementing it
      - Many solutions are introduced (will cover them later)
  - Global Clock Overflow
    - If a small variable (32-bit) is used it can overflow after 2^{32} transaction
      - Using 64-bit counter makes it a theoretical limit