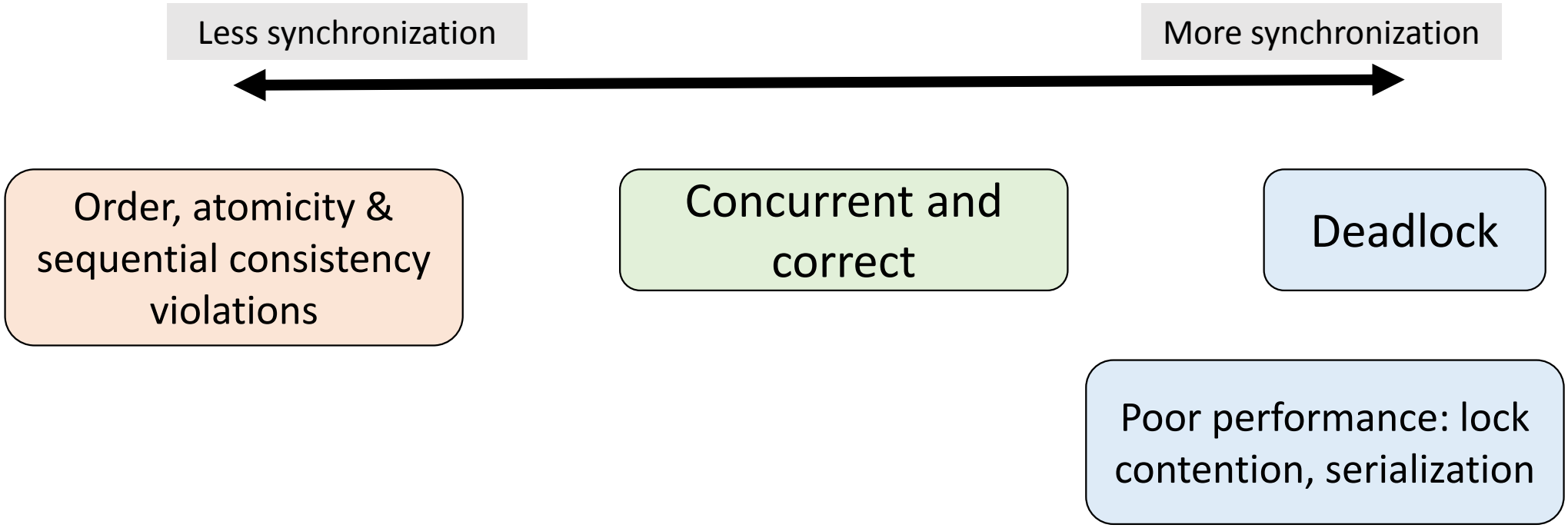
Challenges with Concurrent Programming



Locking relies on programmer conventions!

- If a thread holding a lock is delayed, other contendi
  make progress
- D
- P
- Lo

```
/*
 * When a locked buffer is visible to the I/O layer
 * BH_Launder is set. This means before unlocking
 * we must clear BH_Launder,mb() on alpha and then
 * clear BH_Lock, so no reader can see BH_Launder set
 * on an unlocked buffer and then risk to deadlock.
 */
```

Actual comment from Linux Kernel

Bradley Kuszmaul, and Maurice Herlihy and Nir Shavit

Transactional Memory

- Transaction: A computation sequence that executes ***as if*** without external interference
  - Computation sequence appears indivisible and instantaneous

- Proposed by Lomet [‘77] and Herlihy and Moss [‘93]

Advantages of TM

Programmer says what needs to be atomic

- TM system/runtime implements synchronization

Declarative abstraction

- Programmer says **what work** should be done
- Compare with imperative abstraction
  - Programmer says **how work** should be done

Easy programmability (like coarse-grained locks)

- Performance goal is like fine-grained locks

Basic TM Design

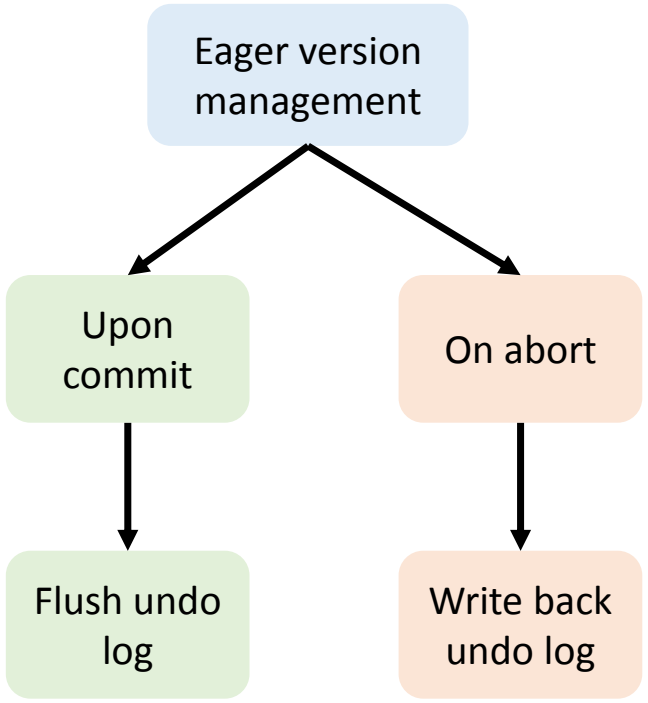
- Transactions are executed **speculatively**
- If the transaction execution completes without a conflict, then the transaction **commits**
  - The updates are made permanent
- If the transaction experiences a conflict, then it **aborts**

Version Management

TMs need to track updates for conflict resolution

Eager

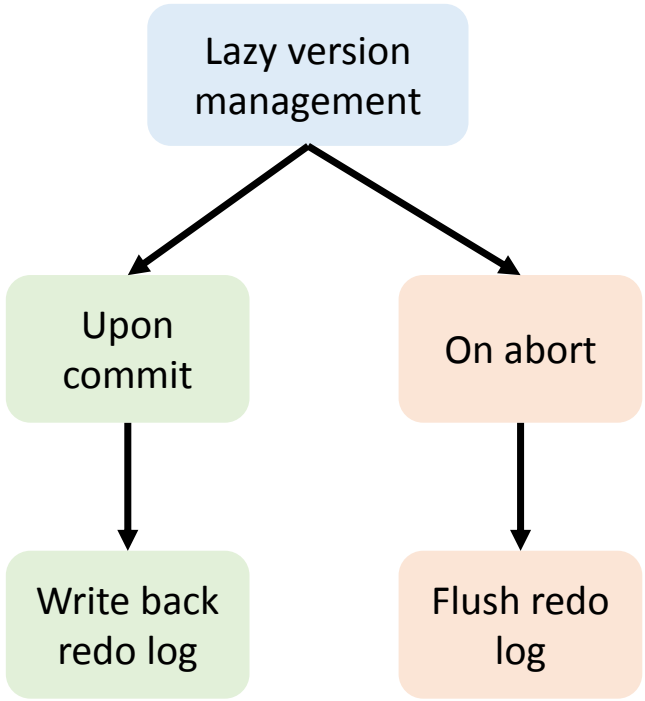
- Tx directly updates data in memory (direct update)
- Maintains an undo log with overwritten values
- Values in the undo log are used to revert updates on an abort



Version Management

Lazy

- Tx updates data in a private redo log
- Updates are made visible at commit (deferred update)
- Tx reads must lookup redo logs
- Discard redo log on an abort



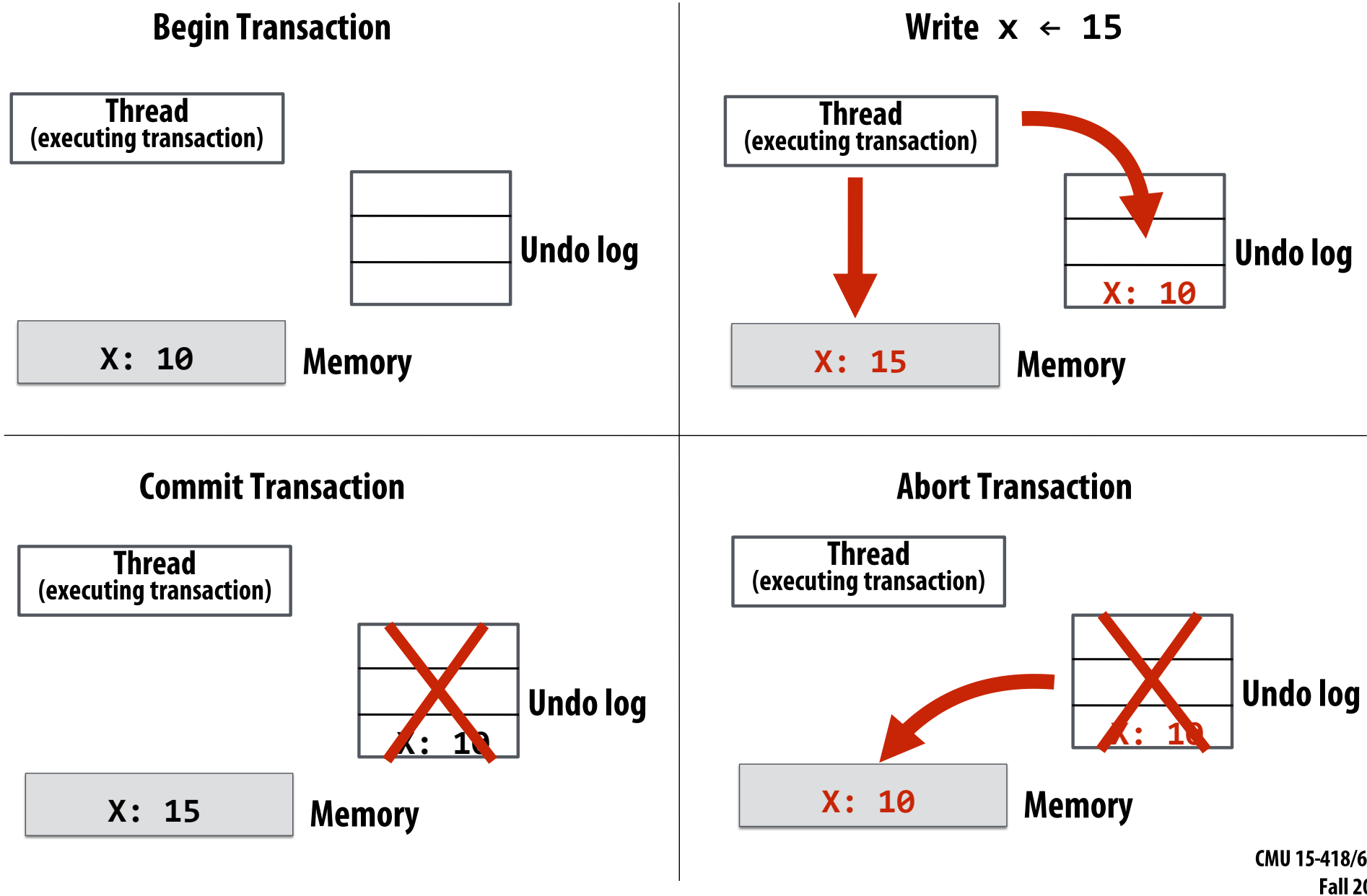
Data versioning

Manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions

1. **Eager versioning (undo-log based)**
2. **Lazy versioning (write-buffer based)**

Eager versioning

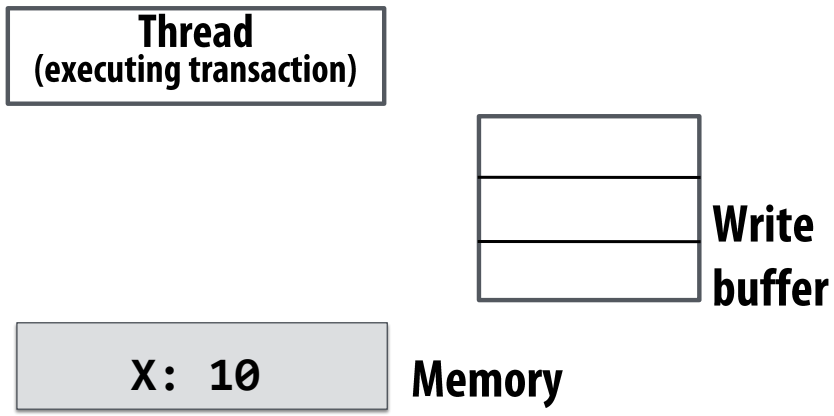
Update memory immediately, maintain "undo log" in case of abort



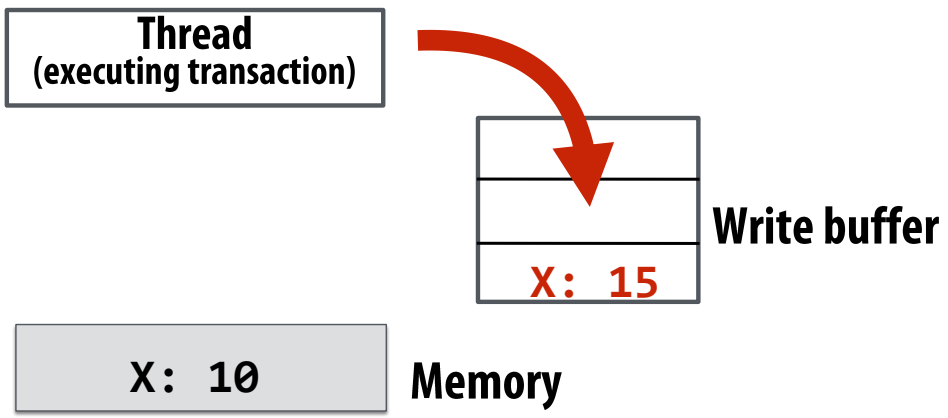
Lazy versioning

Log memory updates in transaction write buffer, flush buffer on commit

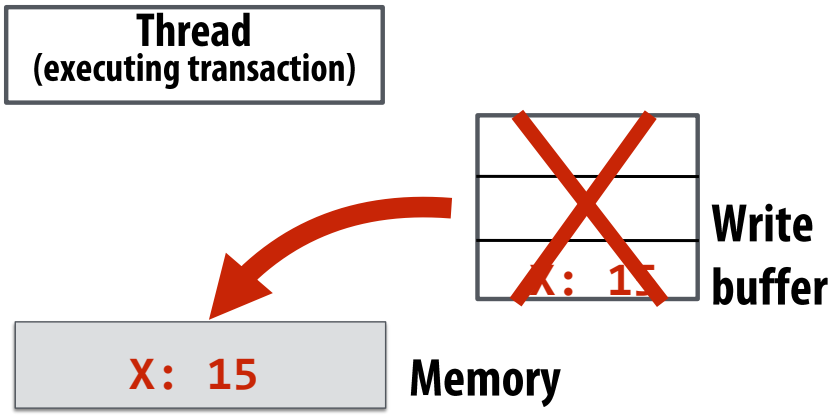
Begin Transaction



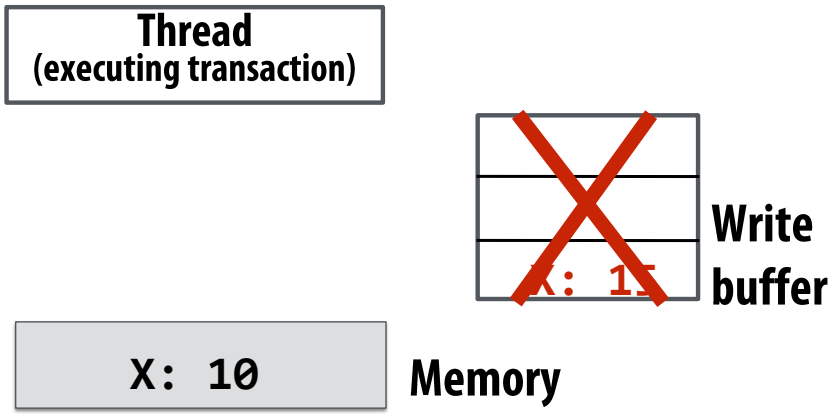
Write x ← 15



Commit Transaction



Abort Transaction



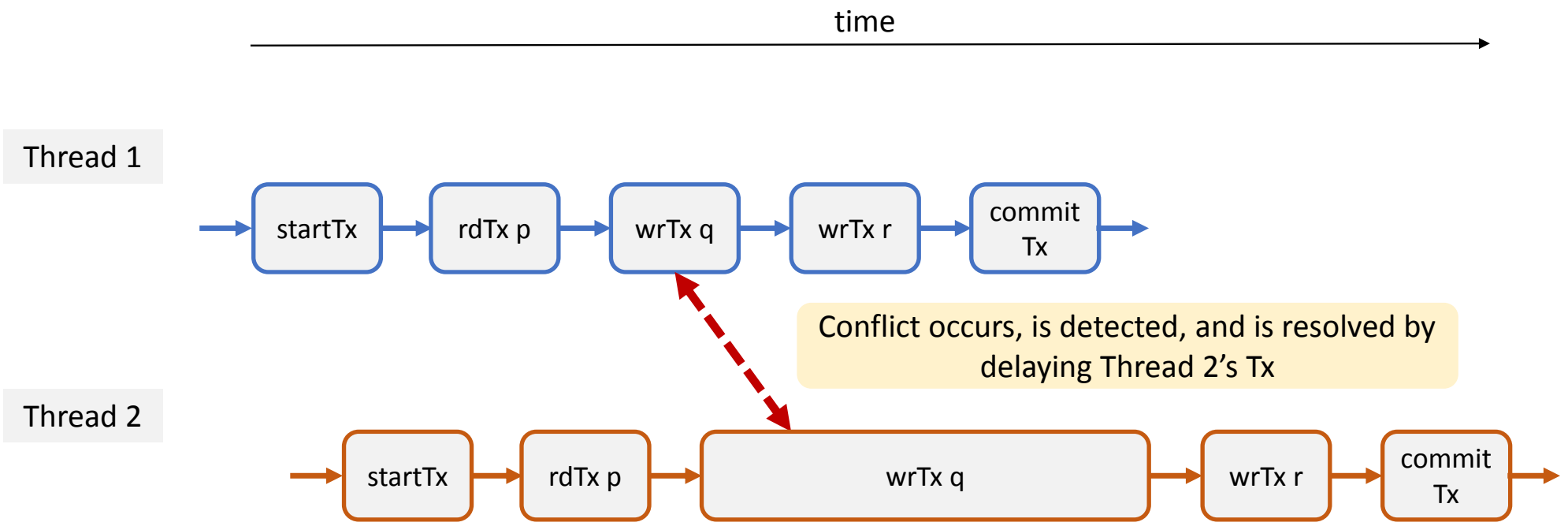
Data versioning

- Manage uncommitted (new) and committed (old) versions of data for concurrent transactions
- Eager versioning (undo-log based)
  - Update memory location directly on write
  - Maintain undo information in a log (incurs per-store overhead)
  - Good: faster commit (data is already in memory)
  - Bad: slower aborts, fault tolerance issues (consider crash in middle of transaction)
- Lazy versioning (write-buffer based)
  - Buffer data in a write buffer until commit
  - Update actual memory location on commit
  - Good: faster abort (just clear log), no fault tolerance issues
  - Bad: slower commits

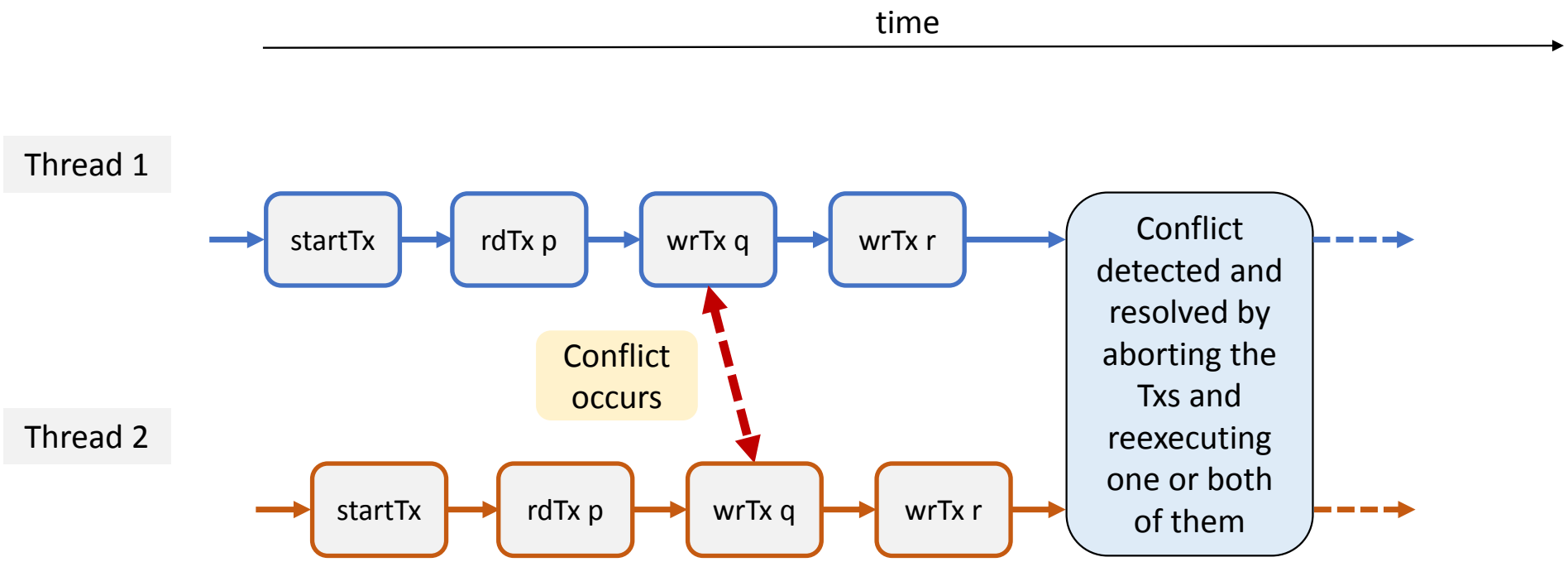
Eager versioning philosophy: write to memory immediately, hoping transaction won't abort (but deal with aborts when you have to)

Lazy versioning philosophy: only write to memory when you have to

Pessimistic Concurrency Control



Optimistic Concurrency Control



Conflict detection

- **Must detect and handle conflicts between transactions**
  - Read-write conflict: transaction A reads address X, which was written to by pending transaction B
  - Write-write conflict: transactions A and B are both pending, and both write to address X
- **System must track a transaction's read set and write set**
  - Read-set: addresses read within the transaction
  - Write-set: addresses written within the transaction

Pessimistic detection

- Check for conflicts during loads or stores
  - A HW implementation will check for conflicts through coherence actions

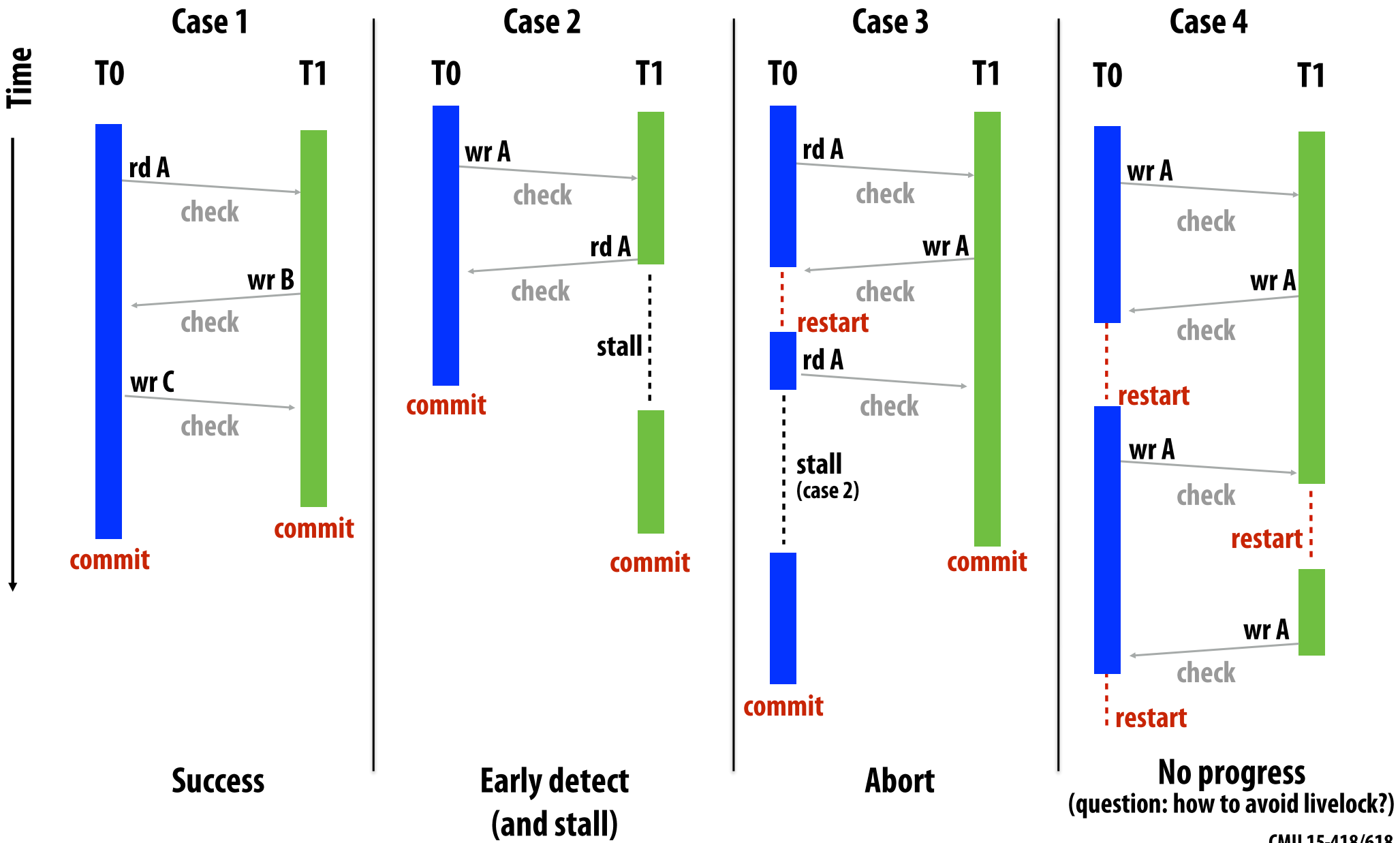
    (will discuss in detail later)

  - Philosophy: "I suspect conflicts might happen, so let's always check to see if one has occurred after each memory operation… if I'm going to have to roll back, might as well do it now to avoid wasted work."

- "Contention manager" decides to stall or abort transaction when a conflict is detected
  - Various priority policies to handle common case fast

Pessimistic detection examples

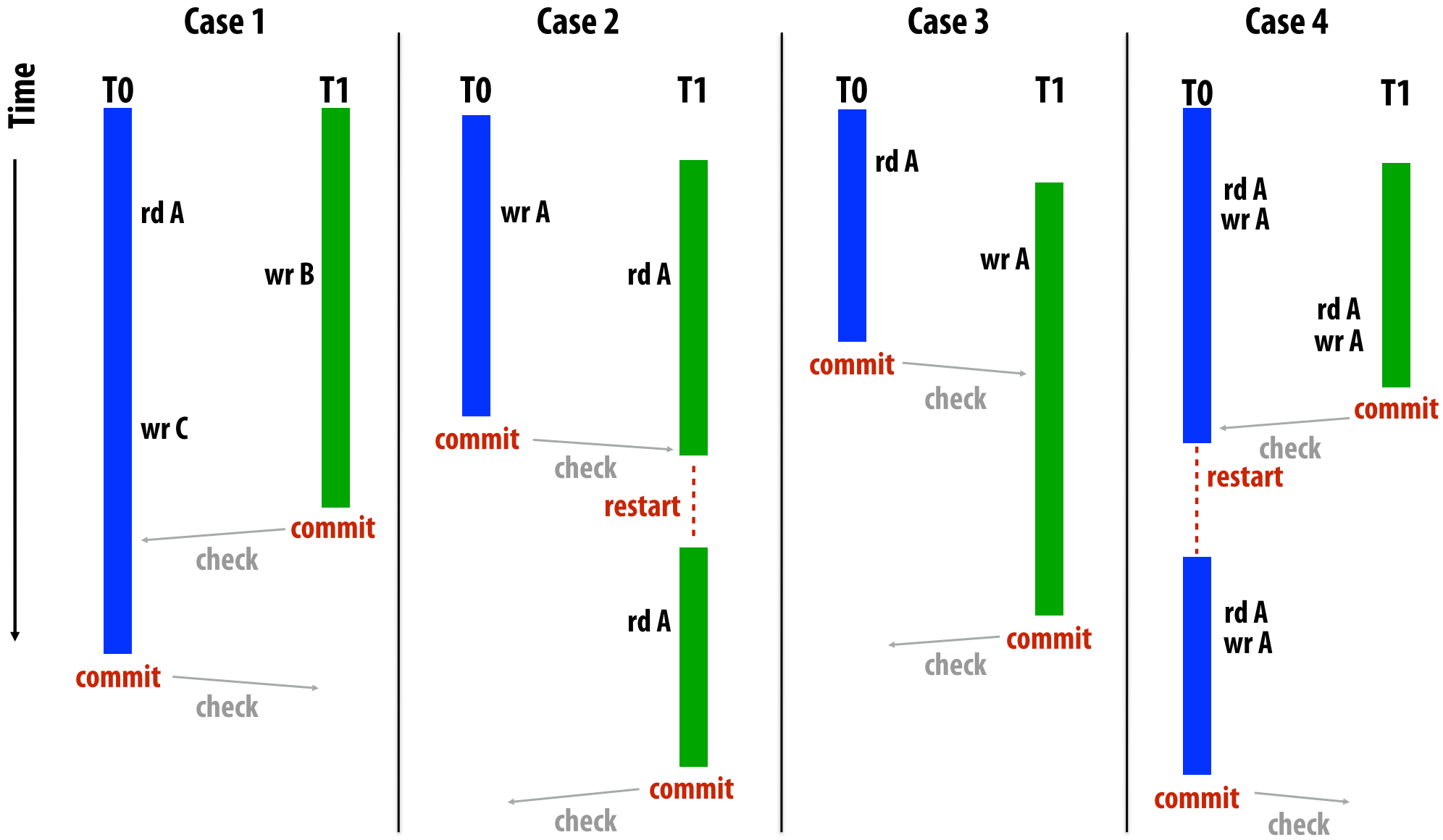
(Note: diagrams assume "aggressive" contention manager on writes: writer wins)



Optimistic detection

- Detect conflicts when a transaction attempts to commit
  - HW: validate write set using coherence actions
    - Get exclusive access for cache lines in write set
  - Intuition: "Let's hope for the best and sort out all the conflicts only when the transaction tries to commit"
- On a conflict, give priority to committing transaction
  - Other transactions may abort later on
  - On conflicts between committing transactions, use contention manager to decide priority
- Note: can use optimistic and pessimistic schemes together
  - Several STM systems use optimistic for reads and pessimistic for writes

Optimistic detection

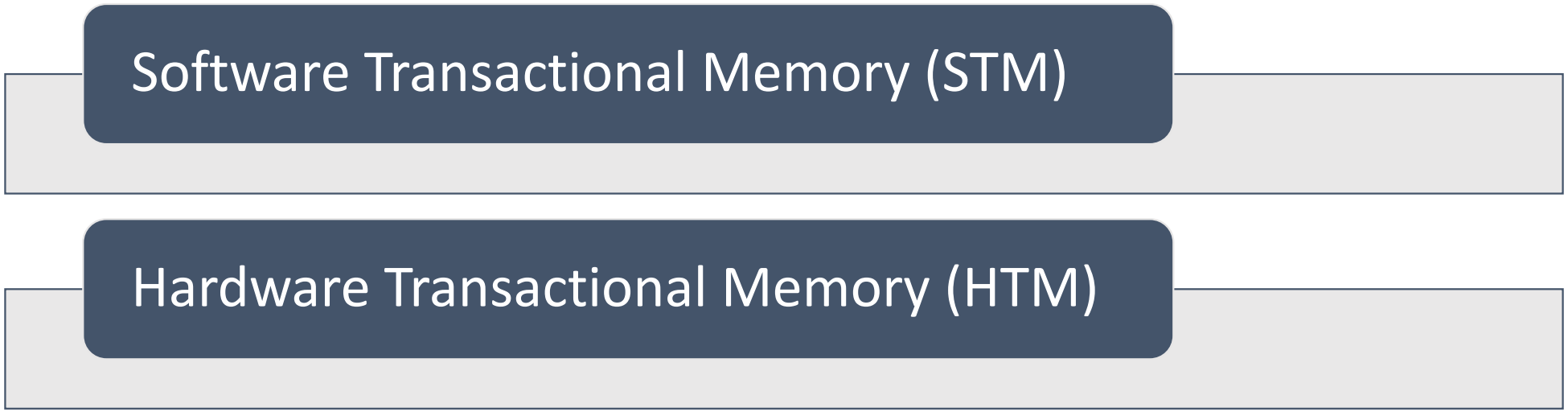


Conflict detection trade-offs

- **Pessimistic conflict detection (a.k.a. "eager")**
  - Good: Detect conflicts early (undo less work, turn some aborts to stalls)
  - Bad: no forward progress guarantees, more aborts in some cases
  - Bad: fine-grained communication (check on each load/store)
  - Bad: detection on critical path

- **Optimistic conflict detection (a.k.a."lazy" or "commit")**
  - Good: forward progress guarantees
  - Good: bulk communication and conflict detection
  - Bad: detects conflicts late, can still have fairness problems

Providing Txs: TM Implementations



STMs vs HTMs

STM

- Supports flexible techniques in TM design
- Easy to integrate STMs with PL runtimes
- Easier to support unbounded Txs with dynamically-sized logs
- More expensive than HTMs

HTM

- Restricted variety of implementations
- Need to adapt existing runtimes to make use of HTM
- Limited by bounded-sized structures like caches
- Better performance than STMs

TM implementation space (examples)

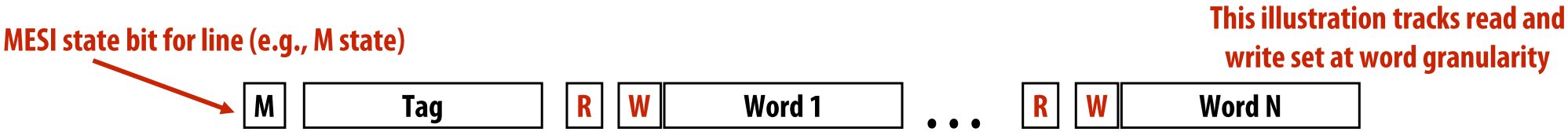
- **Hardware TM systems**
  - Lazy + optimistic: Stanford TCC
  - Lazy + pessimistic: MIT LTM, Intel VTM
  - Eager + pessimistic: Wisconsin LogTM
  - Eager + optimistic: not practical
- **Software TM systems**
  - Lazy + optimistic (rd/wr): Sun TL2
  - Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
  - Eager + optimistic (rd)/pessimistic (wr): Intel STM
  - Eager + pessimistic (rd/wr): Intel STM
- **Optimal design remains an open question**
  - May be different for HW, SW, and hybrid

Hardware transactional memory (HTM)

- Data versioning is implemented in caches
  - Cache the write buffer or the undo log
  - Add new cache line metadata to track transaction read set and write set
- Conflict detection through cache coherence protocol
  - Coherence lookups detect conflicts between transactions
  - Works with snooping and directory coherence
- Note:
  - Register checkpoint must also be taken at transaction begin (to restore execution context state on abort)

HTM design

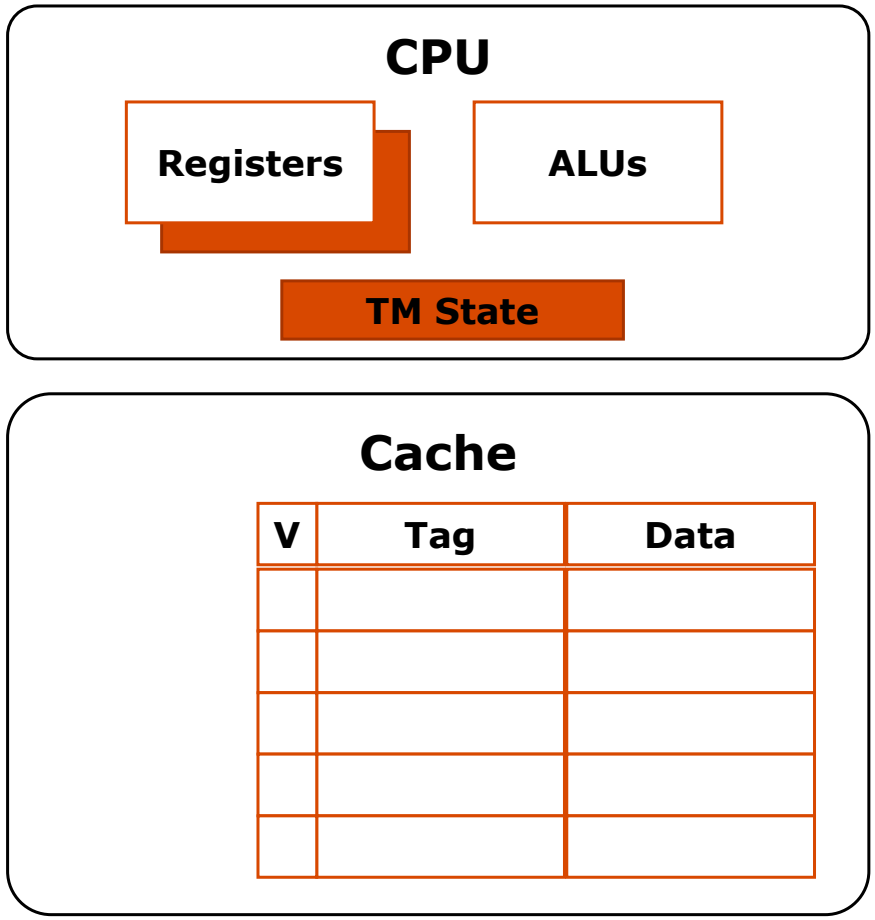
- Cache lines annotated to track read set and write set
  - R bit: indicates data read by transaction (set on loads)
  - W bit: indicates data written by transaction (set on stores)
    - R/W bits can be at word or cache-line granularity
  - R/W bits gang-cleared on transaction commit or abort
  - For eager versioning, need a 2nd cache write for undo log



- Coherence requests check R/W bits to detect conflicts
  - Observing shared request to W-word is a read-write conflict
  - Observing exclusive (intent to write) request to R-word is a write-read conflict
  - Observing exclusive (intent to write) request to W-word is a write-write conflict

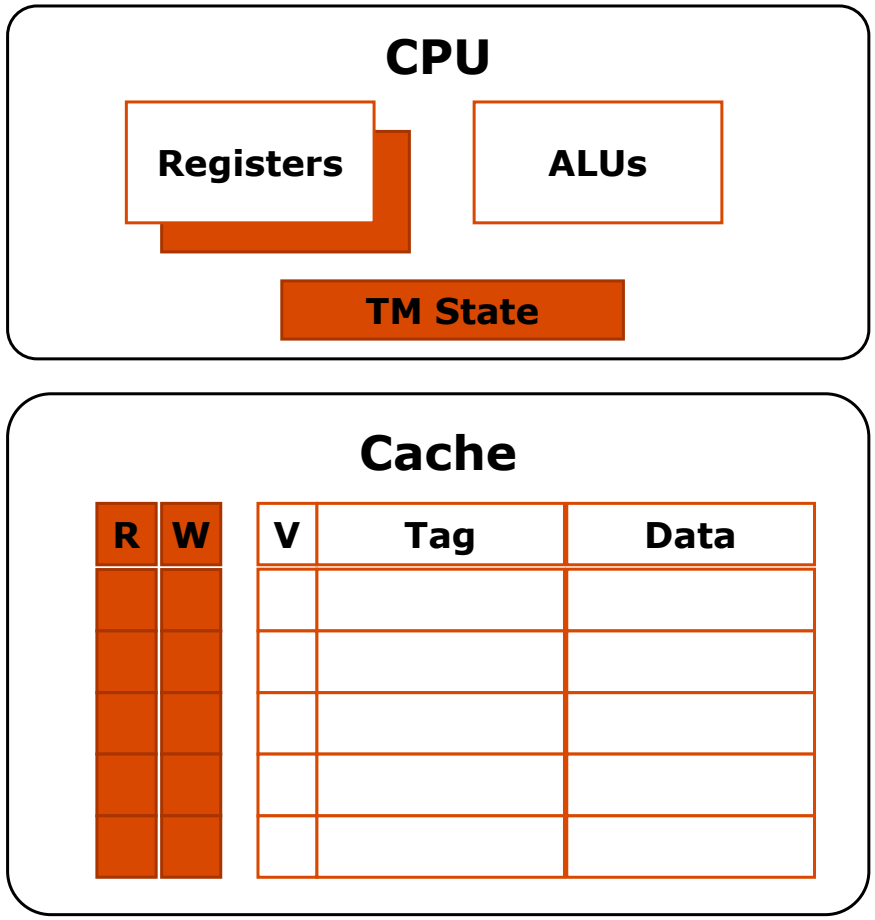
Example HTM implementation: lazy-optimistic

(not Intel's TSX)



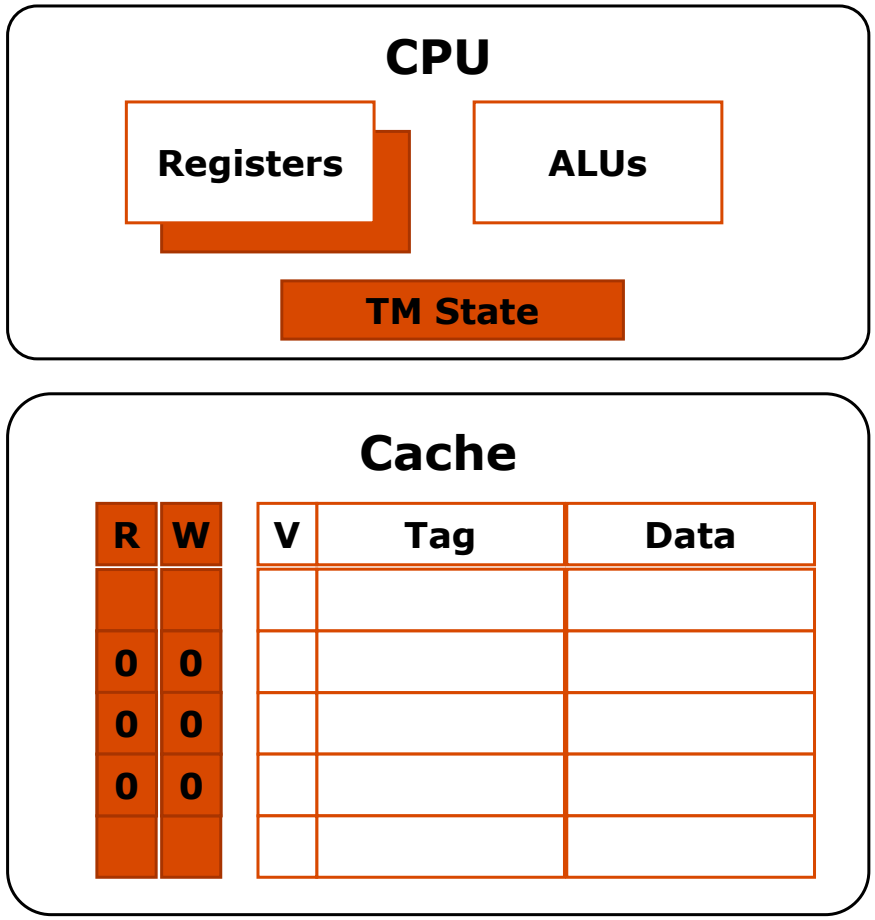
- CPU changes
  - Ability to checkpoint register state (available in many CPUs)
  - TM state registers (status, pointers to abort handlers, . . .)

Example HTM implementation: lazy-optimistic



- Cache changes
  - R bit indicates membership to read set
  - W bit indicates membership to write set

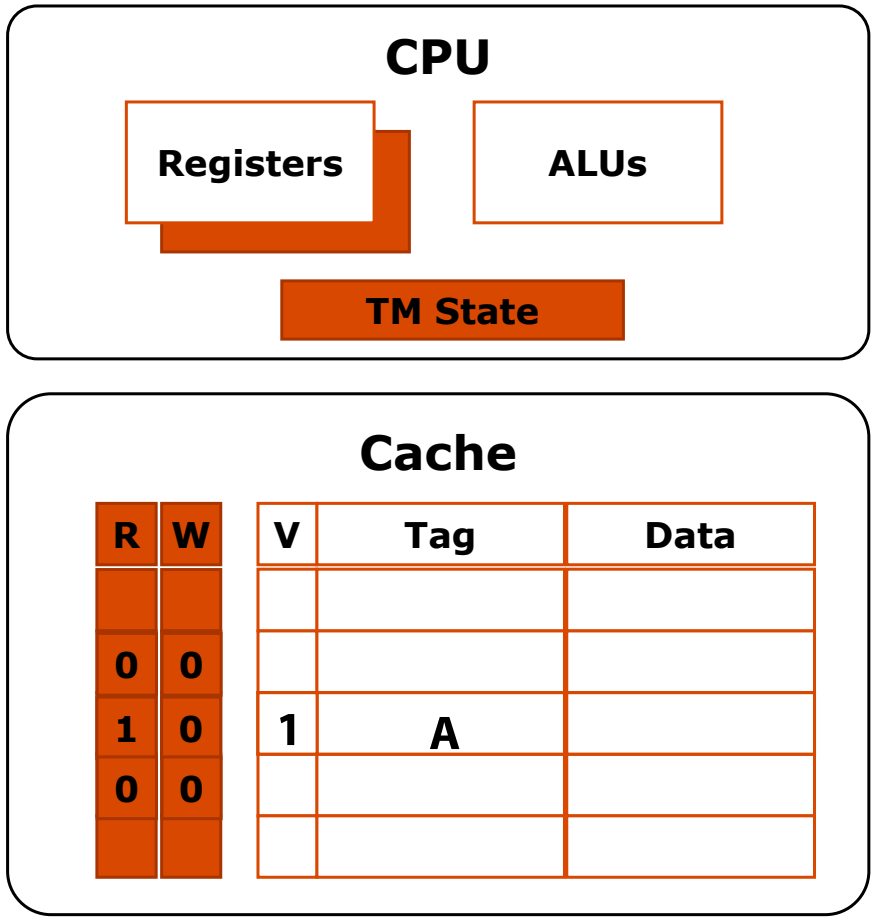
HTM transaction execution




```
Xbegin ⬅
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

- **Transaction begin**
  - Initialize CPU and cache state
  - Take register checkpoint

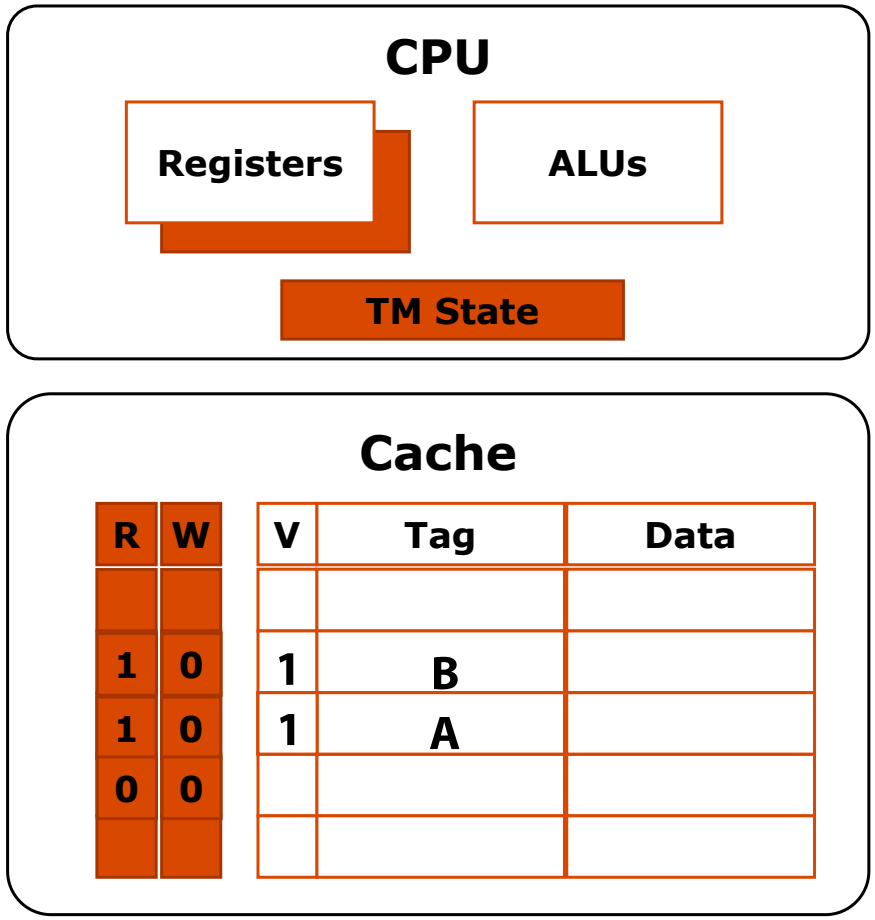
HTM transaction execution




```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

- **Load operation**
  - Serve cache miss if needed
  - Mark data as part of read set

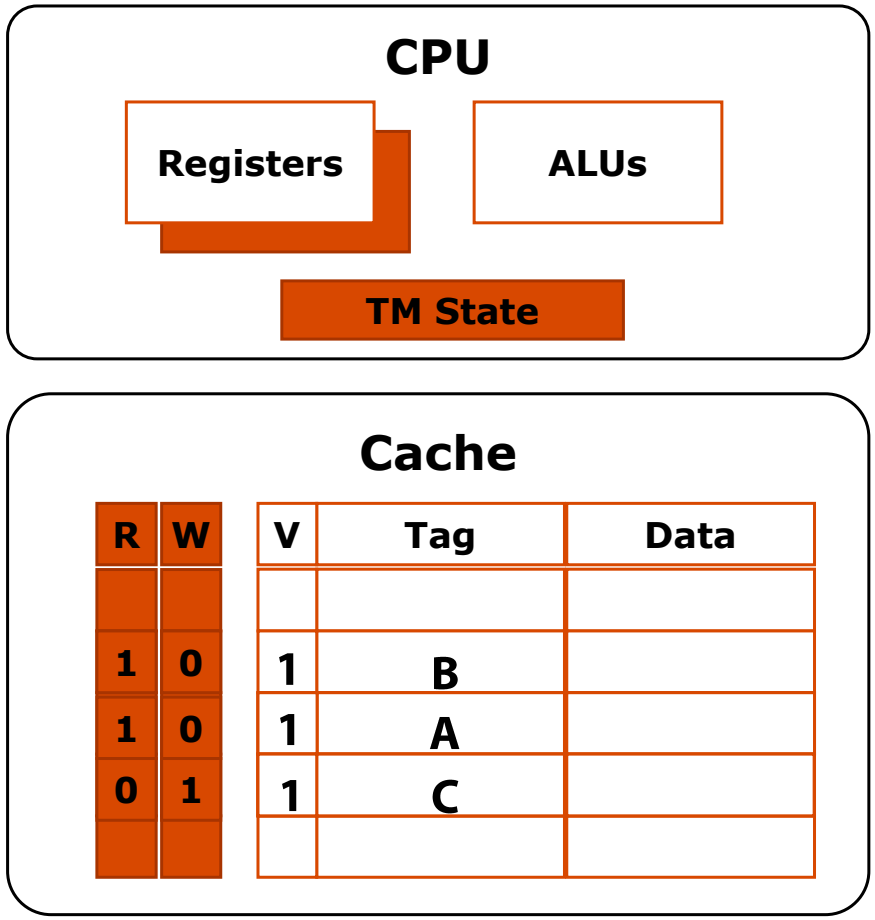
HTM transaction execution




```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

- **Load operation**
  - Serve cache miss if needed
  - Mark data as part of read set

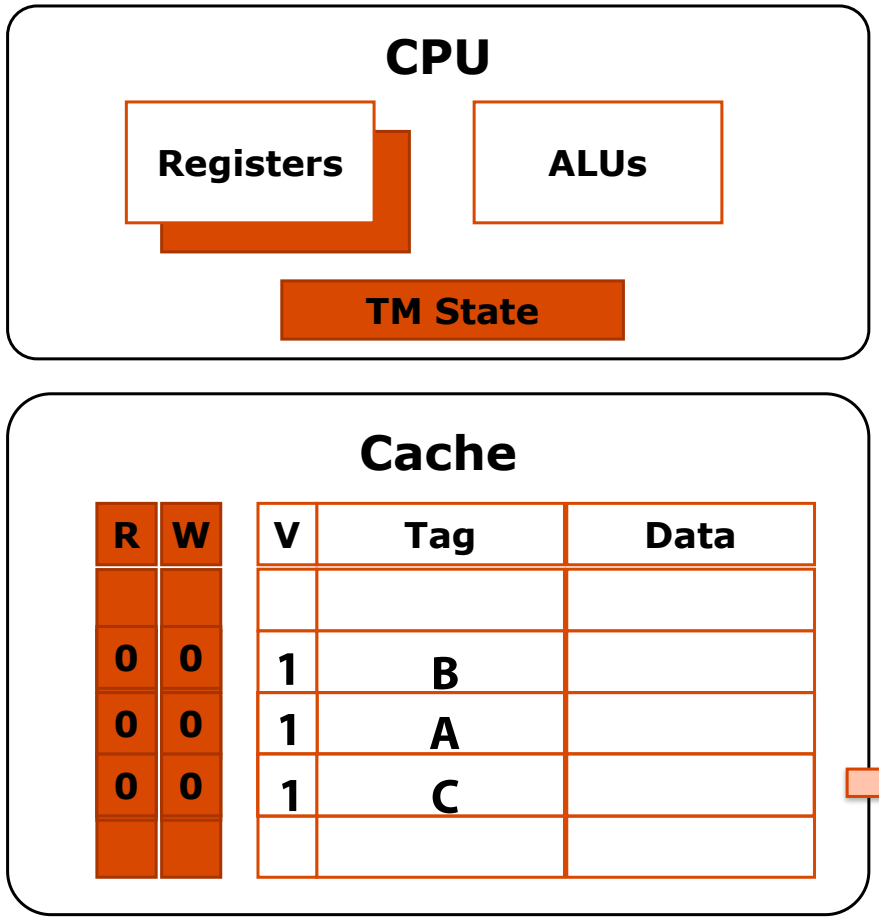
HTM transaction execution




```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

- **Store operation**
  - Service cache miss if needed
  - Mark data as part of write set (note: this is not a load into exclusive state. Why?)

HTM transaction execution: commit



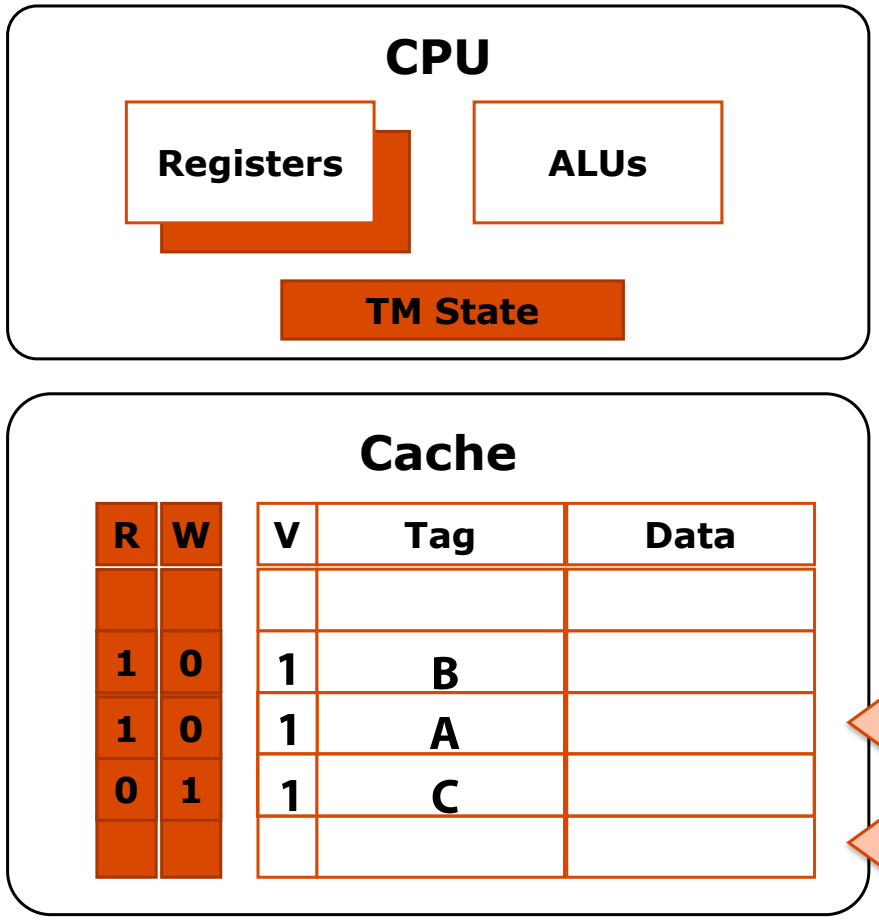

```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

upgradeX C
(result: C is now in exclusive-dirty state)

- Fast two-phase commit
  - Validate: request RdX access to write set lines (if needed)
  - Commit: gang-reset R and W bits, turns write set data to valid (dirty) data

HTM transaction execution: detect/abort

Assume remote processor commits transaction with writes to A and D




```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

coherence requests from another core's commit

(remote core's write of A conflicts with local read of A: triggers abort of pending local transaction)

- **Fast conflict detection and abort**
  - Check: lookup exclusive requests in the read set and write set
  - Abort: invalidate write set, gang-reset R and W bits, restore to register checkpoint

Software Transactional Memory (STM)

Data structures

- Need to maintain per-thread Tx state
- Maintain either redo log or undo log
- Maintain per-Tx read/write sets

- McRT-STM, PPoPP'06
- Bartok-STM, PLDI'06
- JudoSTM, PACT'07
- RingSTM, SPAA'08
- NoRec STM, PPoPP'10
- DeuceSTM, HiPEAC'10
- LarkTM, PPoPP'15

Implementing STM

- Use compilation passes to instrument the program
  - startTx() - Tx entry point (prolog)
  - commitTx() - exit point (epilog)
  - readTx/writeTx - Transactional read/write accesses
- TM runtime tracks memory accesses, detects conflicts, and commits/aborts Txs

```
atomic {
  tmp = x;
  y = tmp + 1;
}
```

```
td = getTxDesc(thr);
startTx(td);
tmp = readTx(&x);
writeTx(&y, tmp+1);
commitTx(td);
```