Programmazione concorrente

Laurea Magistrale in Ingegneria Informatica
Università Tor Vergata
Docente: Romolo Marotta

Transactional Memory

Synchronization approaches:

- **Non-blocking data structures**
- **Locks**
- **Transactional Memory**

Transactional Memory

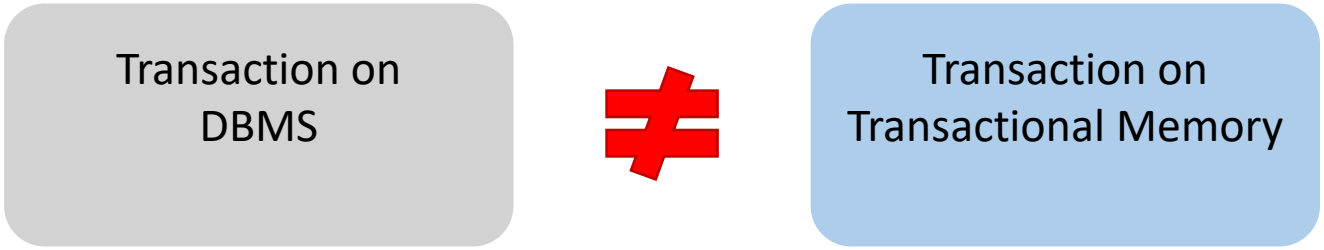
- Why?
  - Fine grain locking (or non-blocking synchronization) can scale but it is hard
  - Locks do not scale in general, but they are hard too:
    - Deadlocks
    - Races (forgotten locks)
    - Do not compose
- Transactions:

```
Begin_transaction
    x.op()
    y.op2(k)
    z.op(j)
End_transaction
```

  - They compose (e.g. nested transactions)
  - Simpler to reason about

Transactions

- Well known in the context of databases
- Conceived integration of transaction in hardware (1993)
- Software implementations (1995-2005)
- Commercial hardware support (2013)

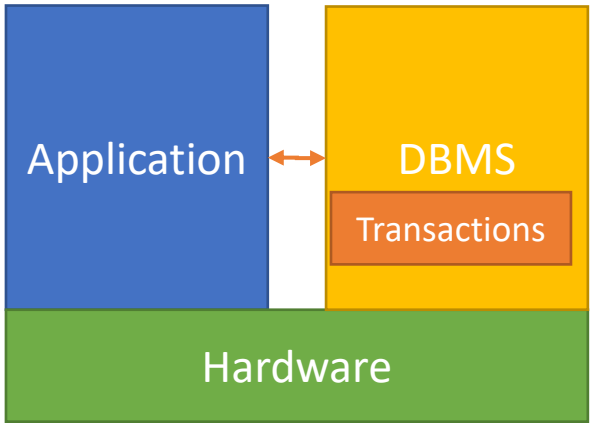


Transactions

Transaction on top of DBMS

≠

Transaction on top of Transactional Memory



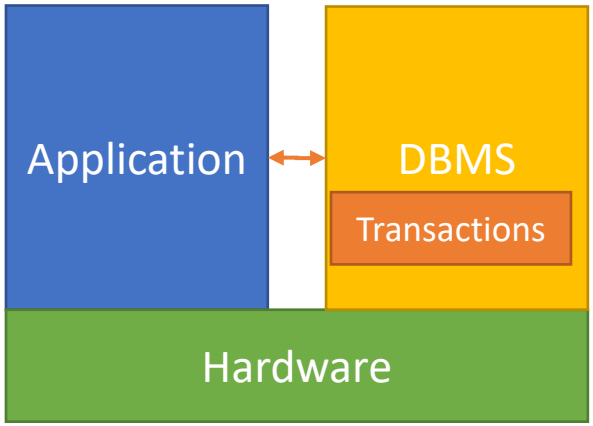

```
x = 2; y = 1
Begin:
    d = x
                                Begin:
                                    y++
                                    x++
                                Commit
    n = y
    write(z, 1/(n-d))
Abort
```

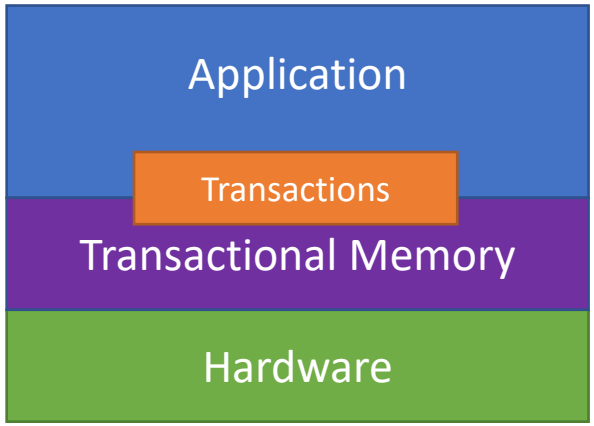
Transactions

Transaction on top of DBMS

≠

Transaction on top of Transactional Memory





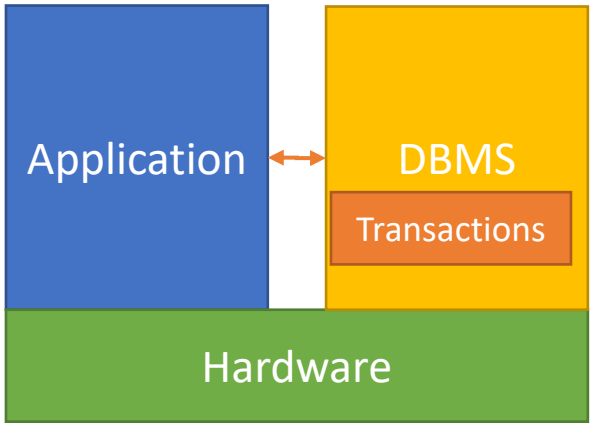

```
x = 2; y = 1
Begin:
    d = x
                                Begin:
                                    y++
                                    x++
                                Commit
    n = y
    write(z, 1/(n-d))
Abort
```

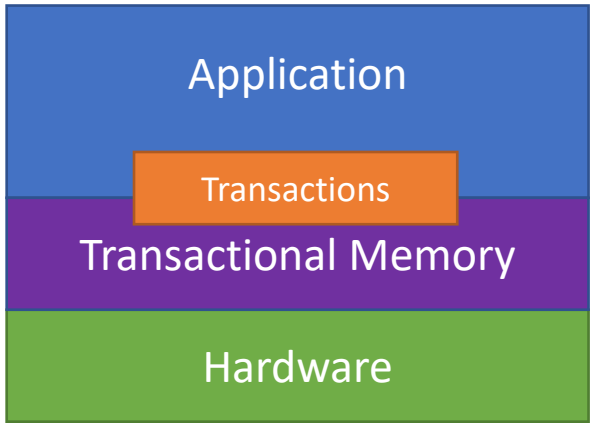
Transactions

Transaction on top of DBMS

≠

Transaction on top of Transactional Memory





Managed by DBMS instead of developers

Float Exceptions are not transparent to developers

```
x = 2; y = 1
B
                Begin:
                  y++
                  x++
                Commit
  n = y
  write(z, 1/(n-d))
Abort
```

Transactions

Transaction on top of DBMS

≠

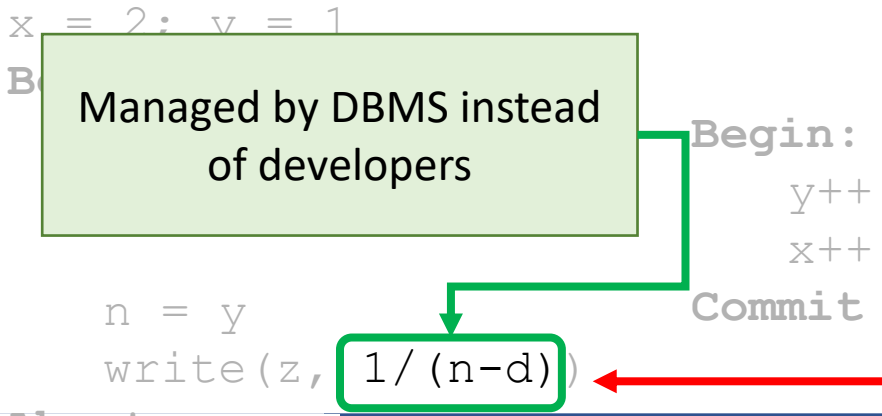
Transaction on top of Transactional Memory

(view) serializability:
Committed transactions see consistent values

Opacity:
Both committed and aborted transactions see consistent values

Managed by DBMS instead of developers

Float Exceptions are not transparent to developers



Histories

- The execution of transaction on a set of objects is modeled by a history
- A history is a sequence of:
  - Operations (e.g., read, write, push, pop …)
  - Commits
  - Aborts
- Two transactions are:
  - **sequential** if one invokes its first operations after the other one commits or aborts
  - **concurrent** otherwise
- A history is:
  - **sequential** if has only sequential transactions
  - **concurrent** otherwise
- Two histories are equivalent if they have the same transactions

Correctness conditions (recall)

- A **concurrent execution** is correct if it is equivalent to a correct **sequential execution**

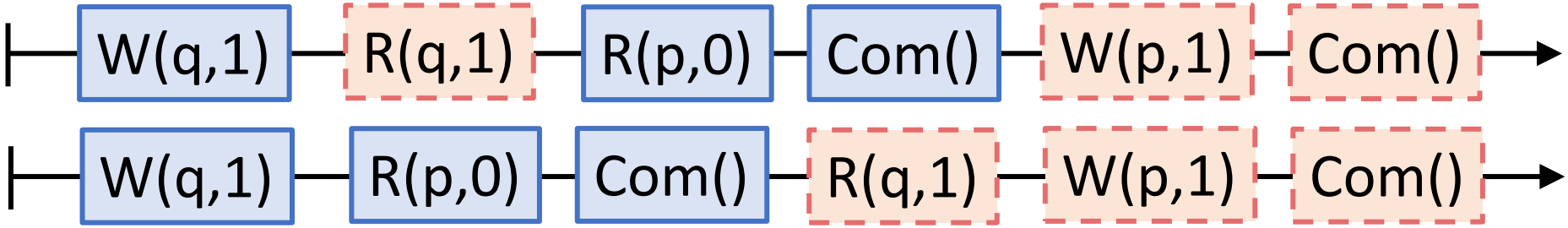
⇒ A **history** is correct if it is **equivalent** to a ~~correct~~ **sequential history** which satisfies a given correctness condition

- A correctness condition specifies the set of histories to be considered as reference

⇒In order to implement correctly a concurrent object wrt a correctness condition, we must guarantee that every possible history on our implementation satisfies the correctness condition

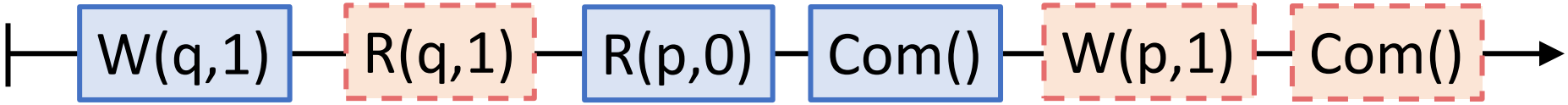
(View) Serializability [Papadimitriou1979]

- A history H of committed transactions is serializable if
  - It is equivalent to a sequential history H'
  - H' is sequential
  - H' is legal, aka every read returns the last written value
- Serializable?

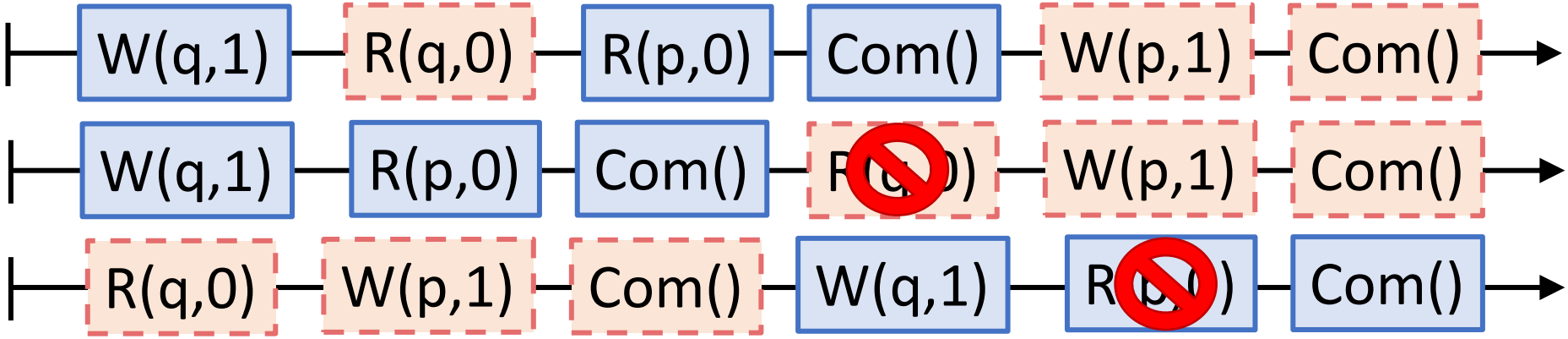


(View) Serializability [Papadimitriou1979]

- A history H of committed transactions is serializable if
  - It is equivalent to a sequential history H'
  - H' is sequential
  - H' is legal, aka every read returns the last written value
- Serializable? Yes!

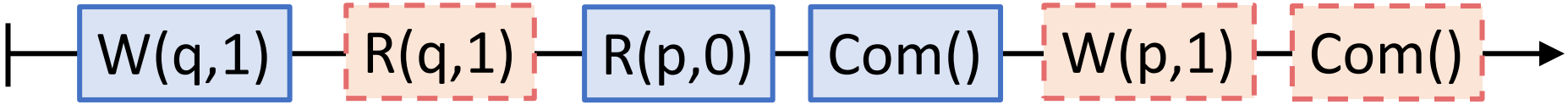


- Serializable?

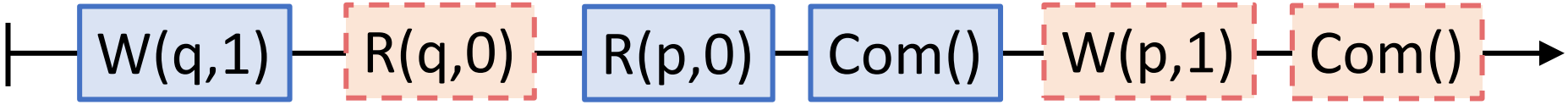


(View) Serializability [Papadimitriou1979]

- A history H of committed transactions is serializable if
  - It is equivalent to a sequential history H'
  - H' is sequential
  - H' is legal, aka every read returns the last written value
- Serializable? Yes!

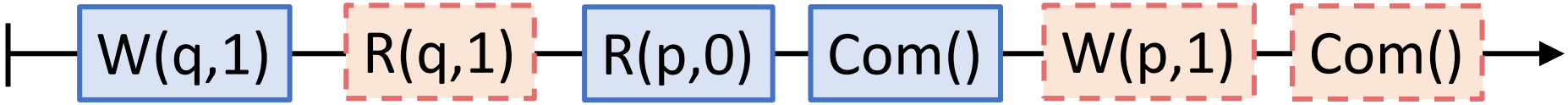


- Serializable? No!

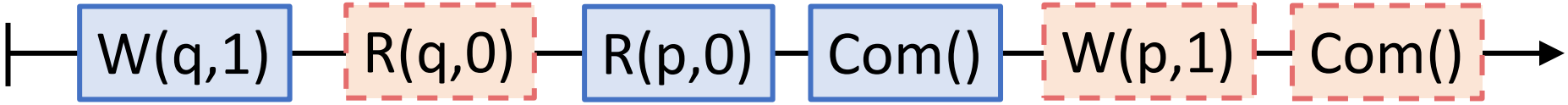


(View) Serializability [Papadimitriou1979]

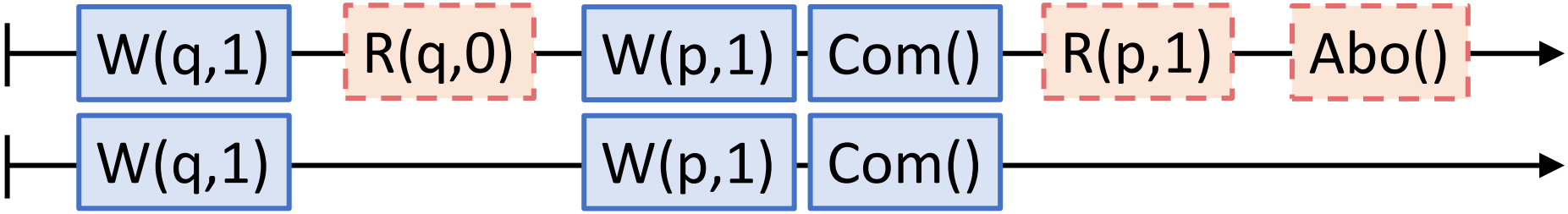
- A history H of committed transactions is serializable if
  - It is equivalent to a sequential history H'
  - H' is sequential
  - H' is legal, aka every read returns the last written value
- Serializable? Yes!



- Serializable? No!

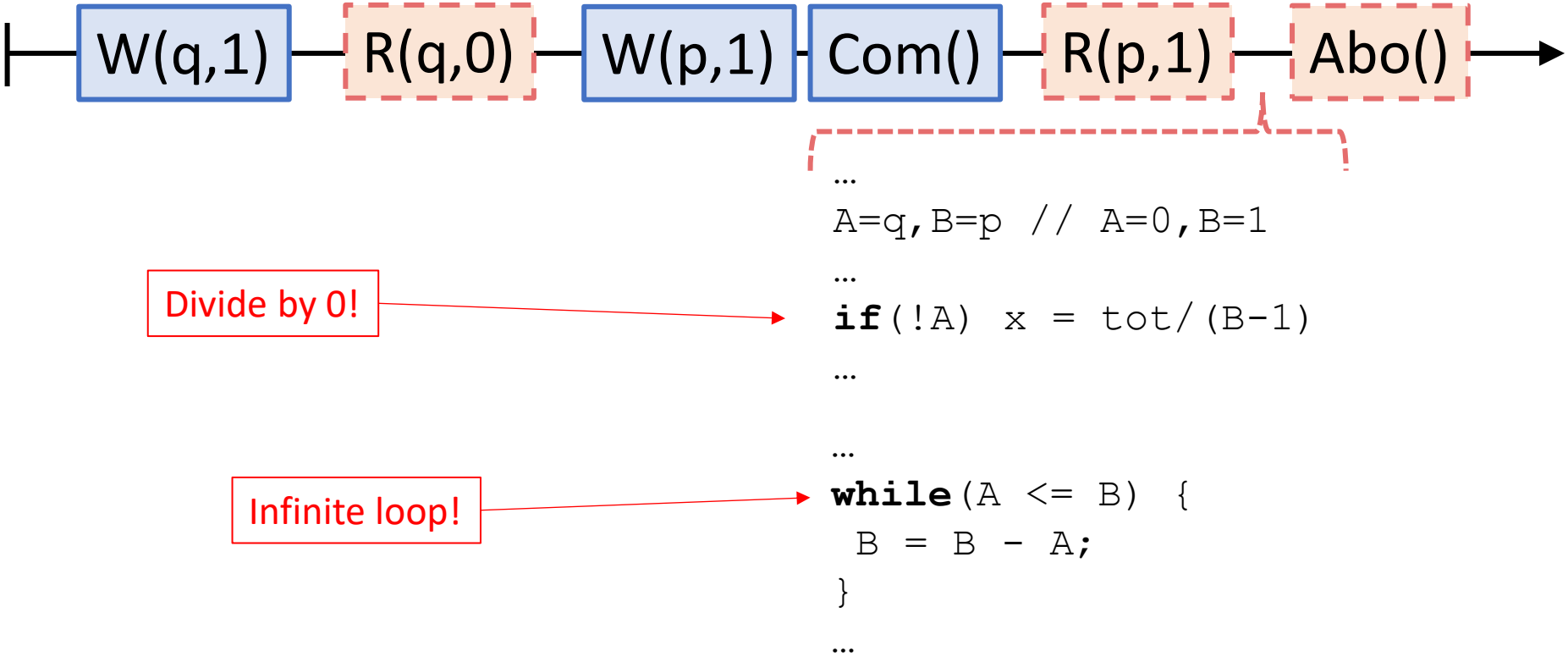


- Serializable? Yes!



(View) Serializability [Papadimitriou1979]

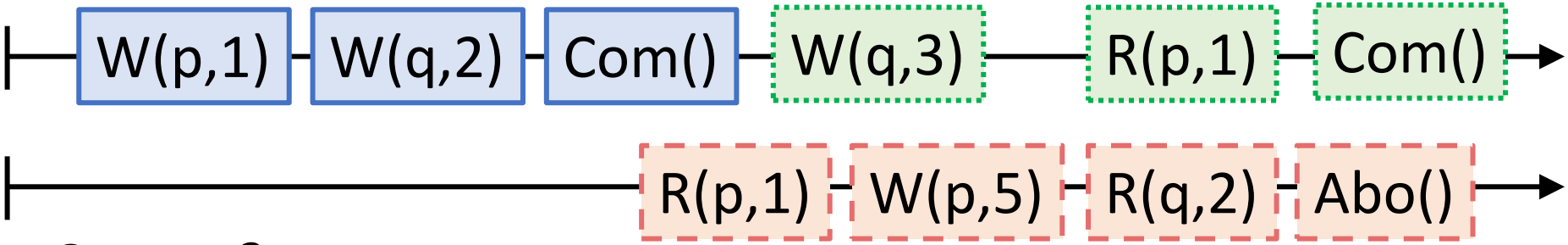
- Serializable? Yes! But, what happens in the case of TM?



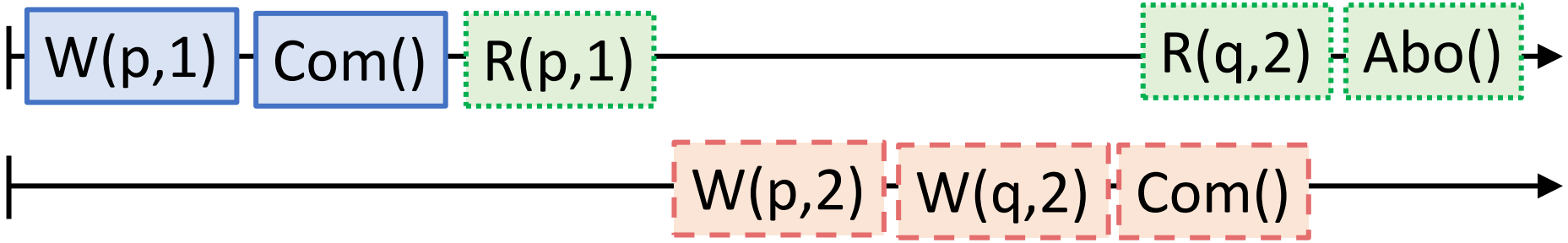
- Could strict serializability be of any help?
  - Serializability + Real-time order
  - It predicates only on committed transactions

Opacity [Guerraoui2008]

- A history H is opaque if
  - It is equivalent to a sequential history H'
  - H' is sequential
  - H' preserves transactions' real-time order
  - H' is legal
- Opaque?



- Opaque?



Transactions

Transaction on top of DBMS

≠

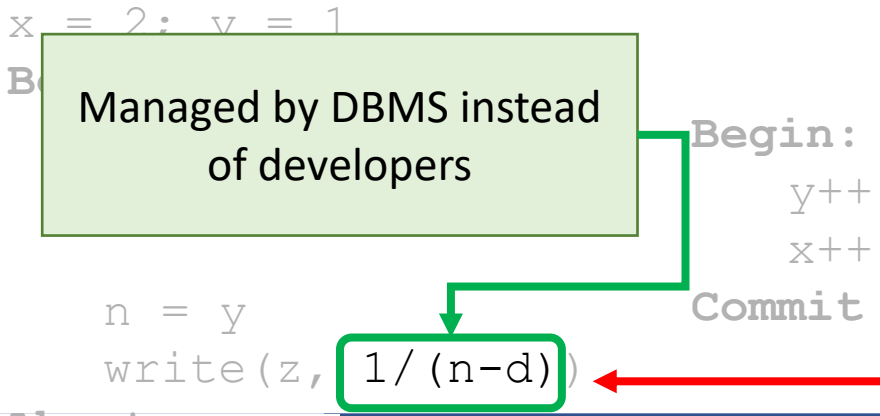
Transaction on top of Transactional Memory

(view) serializability:
Committed transactions see consistent values

Opacity:
Both committed and aborted transactions see consistent values

Managed by DBMS instead of developers

Float Exceptions are not transparent to developers



Transactions

Deadlock or starvation freedom

≠

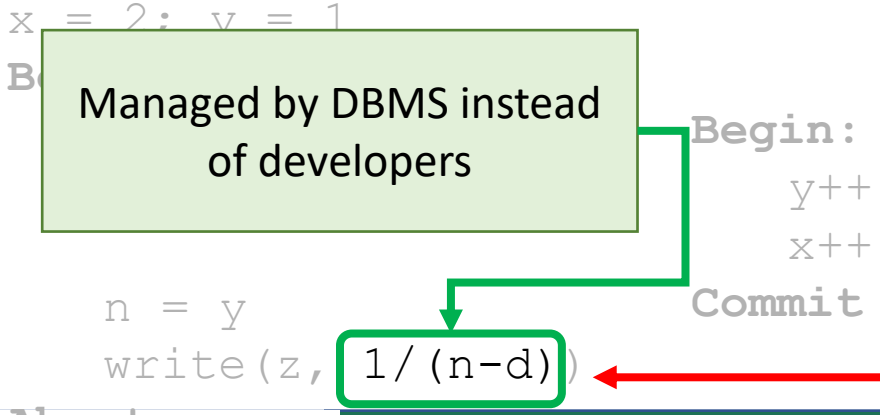
Obstruction freedom

(view) serializability:
Committed transactions see consistent values

Opacity:
Both committed and aborted transactions see consistent values

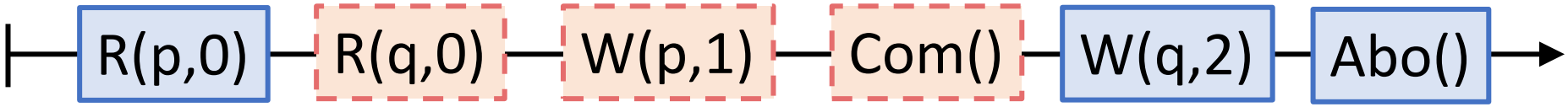
Managed by DBMS instead of developers

Float Exceptions are not transparent to developers



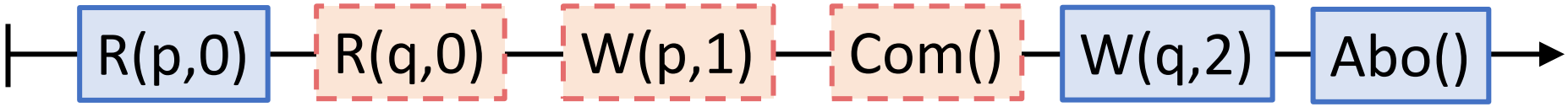
Wait freedom

- Every correct transaction eventually commits
- Finite number of aborts



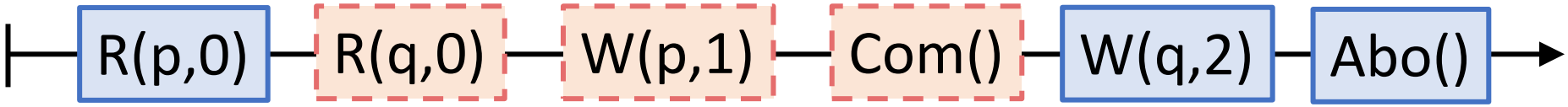
Wait freedom

- Every correct transaction eventually commits
- Finite number of aborts



Wait freedom

- Every correct transaction eventually commits
- Finite number of aborts



IMPOSSIBLE IN AN ASYNCHRONOUS SYSTEM

Obstruction freedom

- Every correct transaction that runs in isolation (without contention) eventually commits
- Abort is unavoidable
- Contention manager can help with contention scenarios
- When a new transaction A creates a conflict with B
  - Aggressive
    - always abort B
  - Backoff
    - B waits an exp. back-off time, then abort A if still conflicting
  - Karma
    - Assign priority to A and B, abort lowest priority, increase priority after abort
  - Greedy
    - Use start time as priority, if Pb < Pa and A is not waiting then B wait, otherwise abort A

Transactions

Deadlock or starvation freedom

≠

Obstruction freedom

(view) serializability:
Committed transactions see consistent values

Opacity:
Both committed and aborted transactions see consistent values

```
x = 2; y = 1
Begin:
    d = x

                    Begin:
                        y++
                        x++
                    Commit

    n = y
    write(z, 1/(n-d))
Abort
```

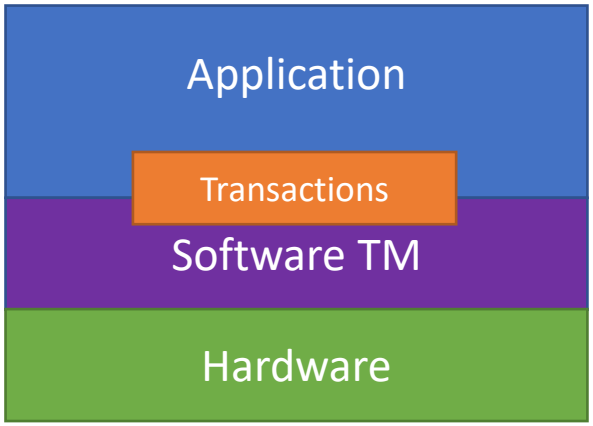
```
x = 2; y = 1
Begin:
    d = x

                    Begin:
                        y++
                        x++
                    Commit

Abort
```

Software Transactional Memory

- DSTM
- JVSTM
- RSTM
- TL2
- TinySTM
- SwissTM
- McRT-STM
- Bartok-STM
- NOrec
- LSA
- E-STM
- SXM
- ASTM
- WSTM
- PhTM



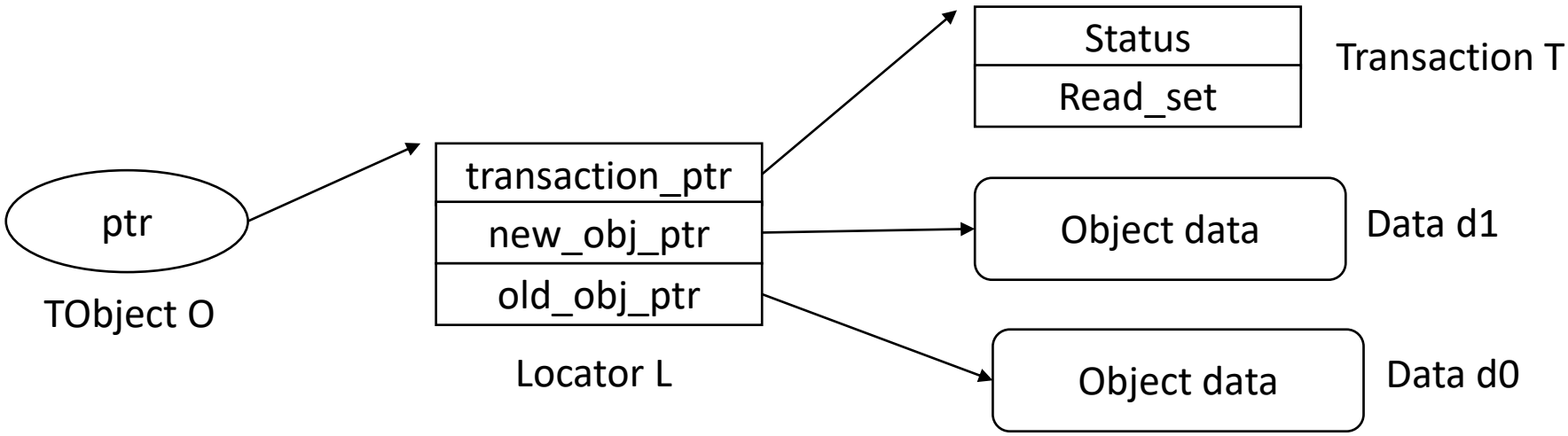
DSTM [Hearlihy2003]

- Obstruction freedom + contention manager
- It works at object granularity
  - Transactions open objects in READ/WRITE mode to apply an operation
  - Conflicts are detected when opening objects
- A conflicting write makes one of the two conflicting transaction abort via contention manager (**killer write**)
- A read requires that all already-read objects are still the most recently committed version (**careful read**)
- Validate all objects read upon commit

DSTM [Hearlihy2003]

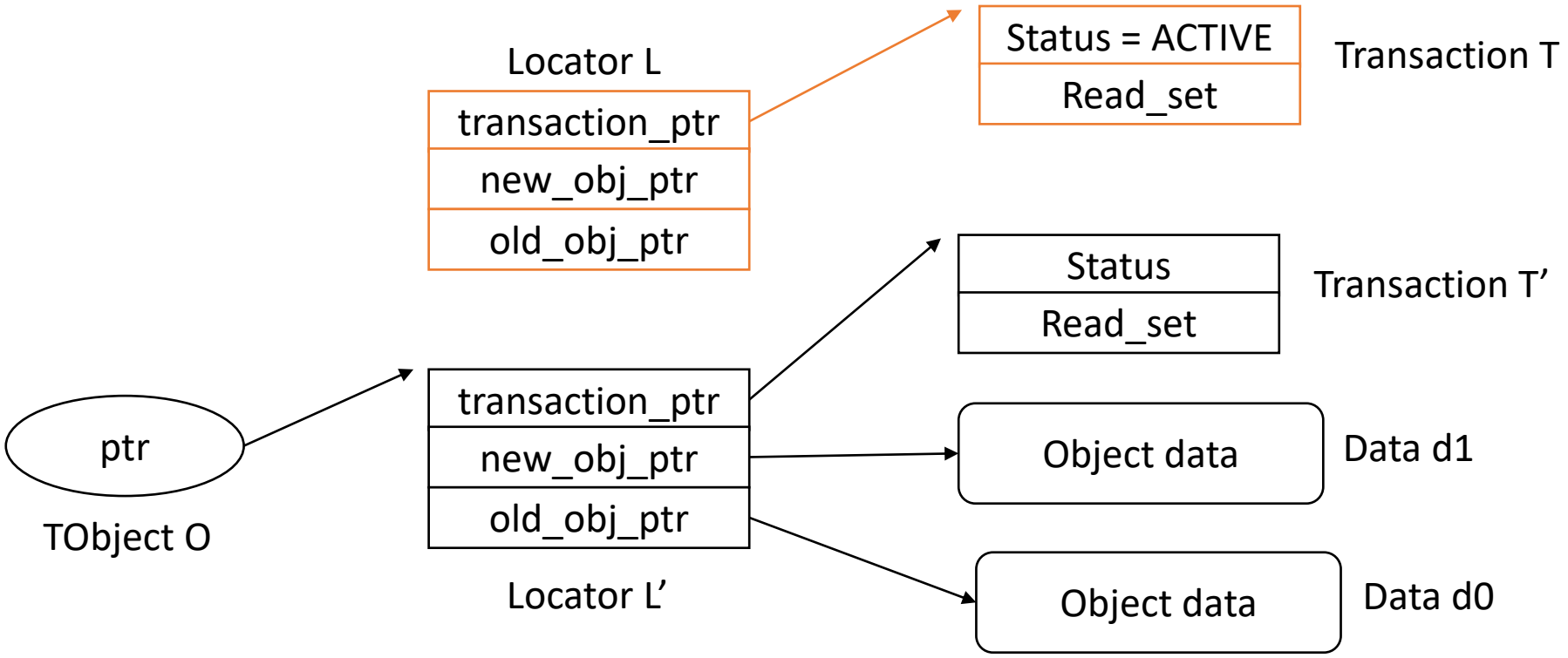
- Transactions have:
  - A status
    - Committed
    - Active
    - Aborted
  - Collection of objects opened in READ mode
- Objects are incapsulated within a Transactional Object which keeps references to
  - Transaction currently manipulating the object in WRITE mode
  - Current and tentative versions of the object

with an intermediate objected called Locator



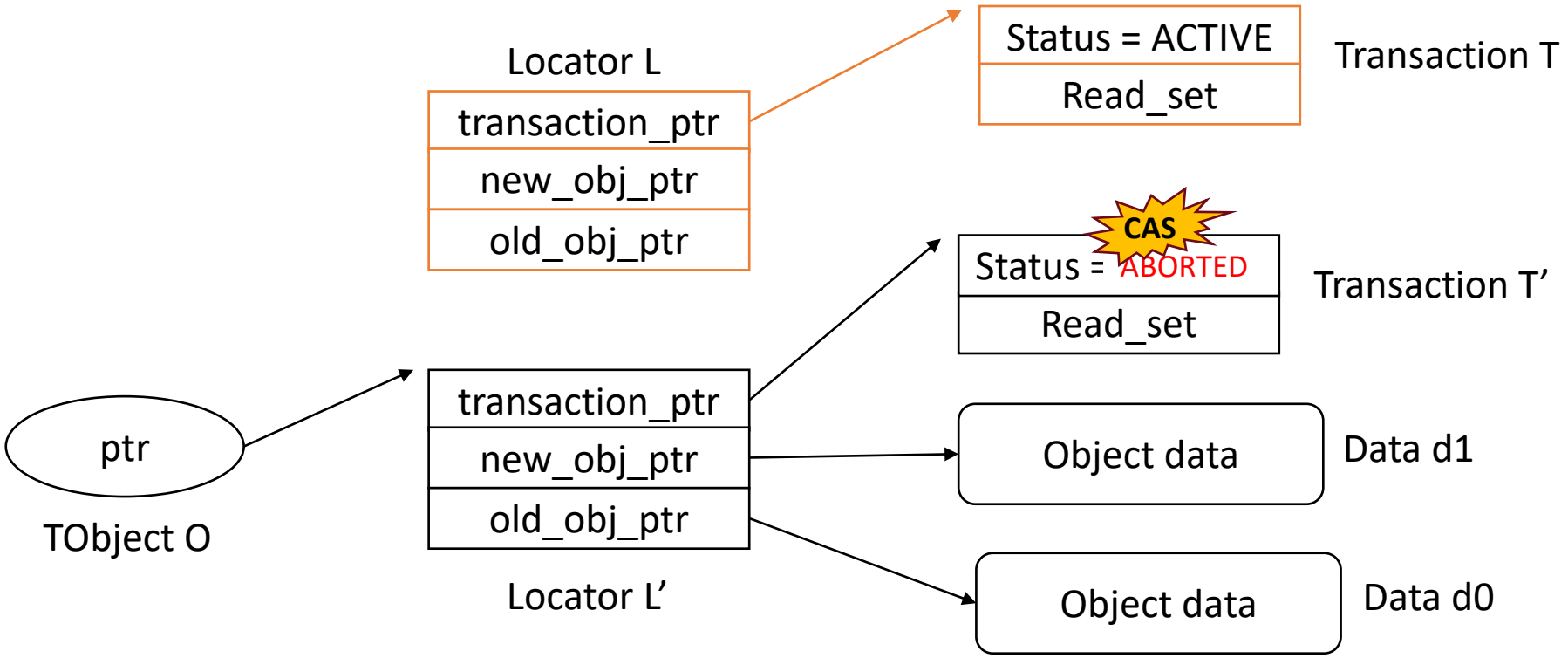
DSTM – First open in WRITE mode [Hearlihy2003]

- **T** is the current transaction, whose status is **ACTIVE**
- **T** allocates a new Locator **L**
- **T** accesses to current locator **L'** of **O** to retrieve last transaction **T'** that executed the last open in WRITE mode



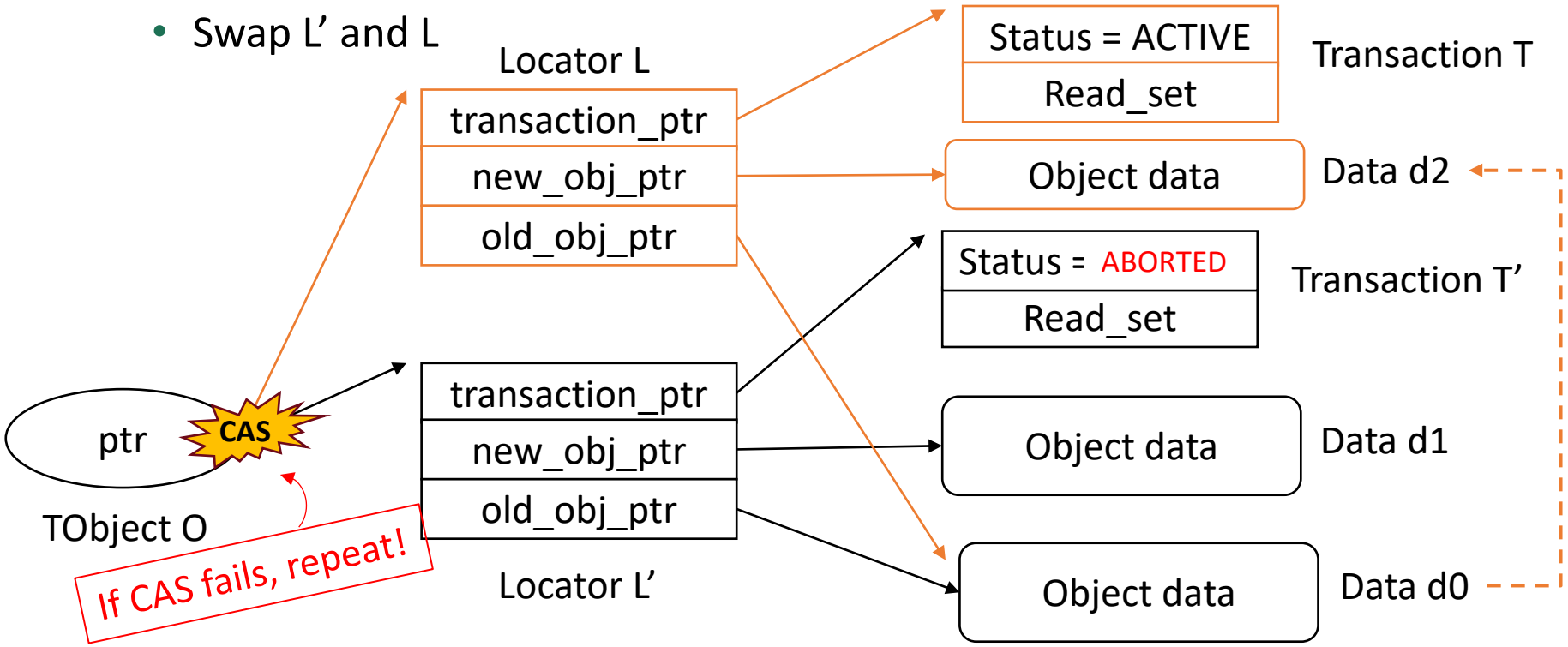
DSTM – First open in WRITE mode [Hearlihy2003]

- **T** behaves accordingly to **T' status**
  - **ACTIVE: T** calls the contention manager
    - **T** waits a back-off time
    - **T** makes **T'** abort via **Compare&Swap**



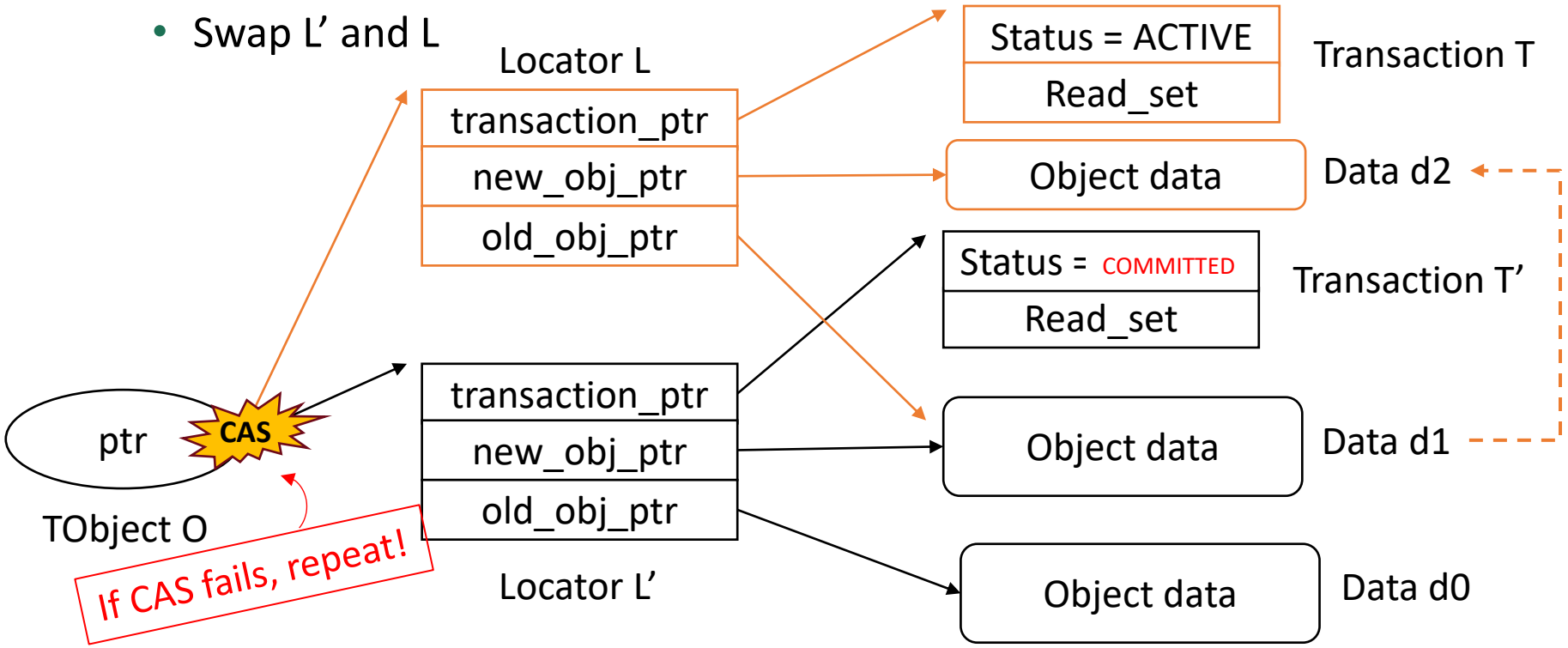
DSTM – First open in WRITE mode [Hearlihy2003]

- **T** behaves accordingly to **T' status**
  - **ABORTED:**
    - **T** use **L'.old_obj_ptr** to get current version of **O**
    - **L.old_obj_ptr = L'.old_obj_ptr**
    - **L.new_obj_ptr = CLONE(L'.old_obj_ptr)**
    - Swap L' and L



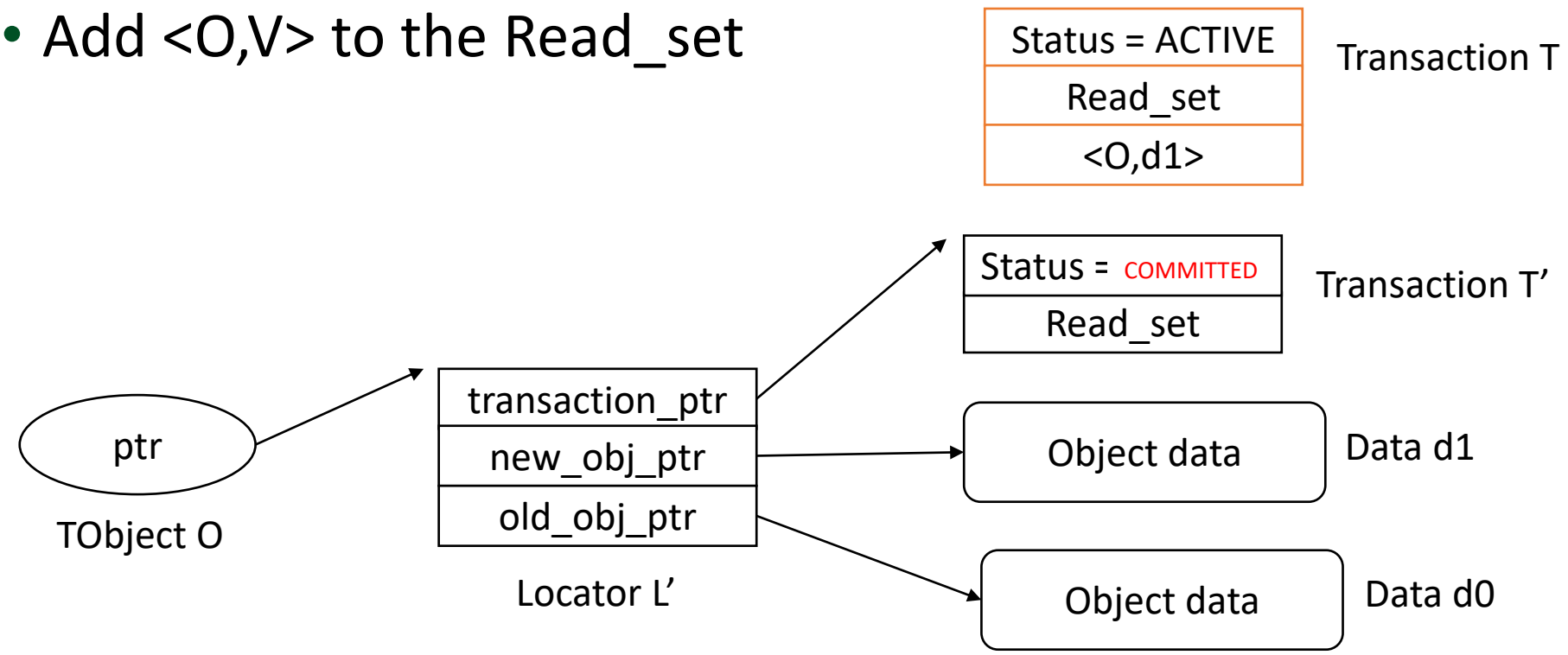
DSTM – First open in WRITE mode [Hearlihy2003]

- **T** behaves accordingly to **T' status**
  - **COMMITTED:**
    - **T** use **L'.old_obj_ptr** to get current version of **O**
    - **L.old_obj_ptr = L'.new_obj_ptr**
    - **L.new_obj_ptr = CLONE(L'.new_obj_ptr)**
    - Swap L' and L



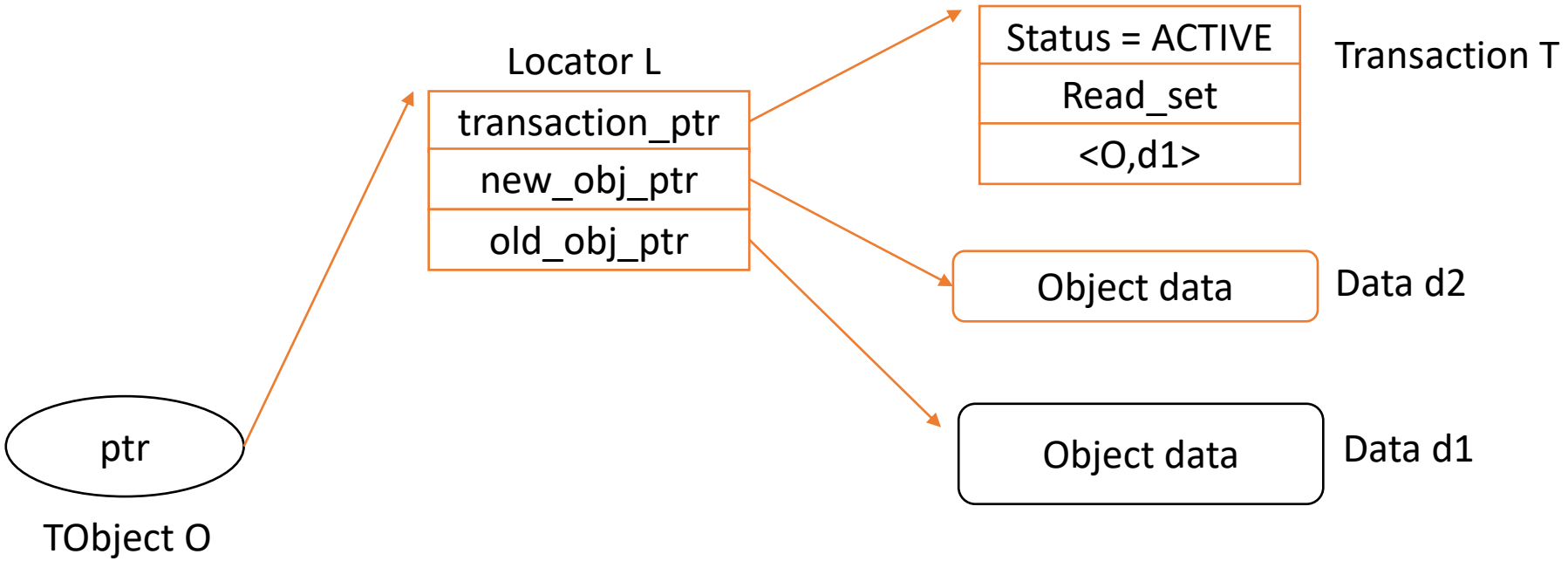
DSTM – First open in READ mode [Hearlihy2003]

- Validate Read_set (see later)
- Fetch current **committed** version **V** via current locator
  - New_obj_ptr if T' is committed
  - Old_obj_ptr otherwise
- Add <O,V> to the Read_set



DSTM – Already opened objects [Hearlihy2003]

- Already opened in READ mode:
  - Retrieve **V** from the Read_set
- Already opened in WRITE mode:
  - Retrieve **V** from the current locator



DSTM – Commit [Hearlihy2003]

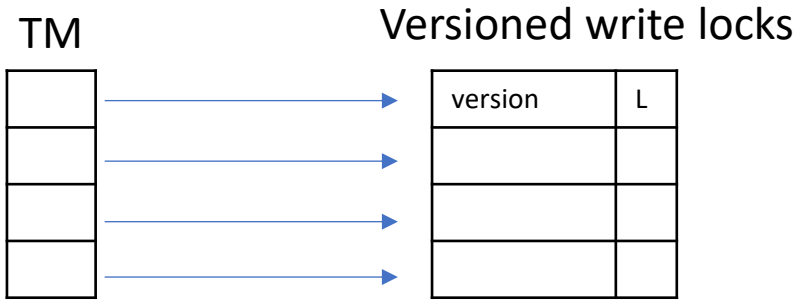
1. Validate the transaction
   - Transaction aborts on WRITE/WRITE conflicts
     - No need to validate WRITE upon commit
   - Validate Read_set
     - For each pair <O,V> check that V is still the most recent committed version
   - Read_set validation is non atomic
     - Check the status is still ACTIVE
2. **If** OK Change status
   then from **ACTIVE** to **COMMITTED**
   else from **ACTIVE** to **ABORTED**
   - Individual CAS

DSTM – Final remarks [Hearlihy2003]

- Read-only transactions do not need any ATOMIC instruction for each read
- Committed transactions appear to take effect when the transition ACTIVE->COMMITTED occurs
  - Linearizable/Strict serializable
- Why careful read (validation at each read)?
- Obstruction freedom
  - Transactions abort iff conflicts occur

Transational Locking 2 [Dice2006]

- Word-based STM
  - Each transactional memory location is associated with a **versioned write lock** <version,is_locked>



- Exploits a Global Version Clock (GVC) to quickly detect updates (it increases before a write-transaction commits)
- Transactions keep track of
  - GVC
  - Read set
  - Write set

Transational Locking 2 [Dice2006]

- BEGIN:
  - Sample GVC and store it in a transaction(thread)-local variable RV
- WRITE(m,v) operation:
  - Add <m,v> to the write set
- READ(m)(v) operation:
  - **IF** m in write set **THEN** return the associated v
  - **ELSE**
    - Load the versioned lock <version,locked> associated to m
      - IF locked or version > RV abort
    - Load v from m
    - IF locked or version > RV abort
    - Add <m> to the readset

Transational Locking 2 [Dice2006]

- COMMIT:
  - For each m in the write set acquire the related versioned lock
    - If acquisition fails abort
  - Increment GVC via Add&Fetch obtaining WV
  - **IF WV != RV+1**
    - Validate the read set (abort if locked or version > RV)
  - Store each value in the write set
  - Release each versioned lock by using WV as version

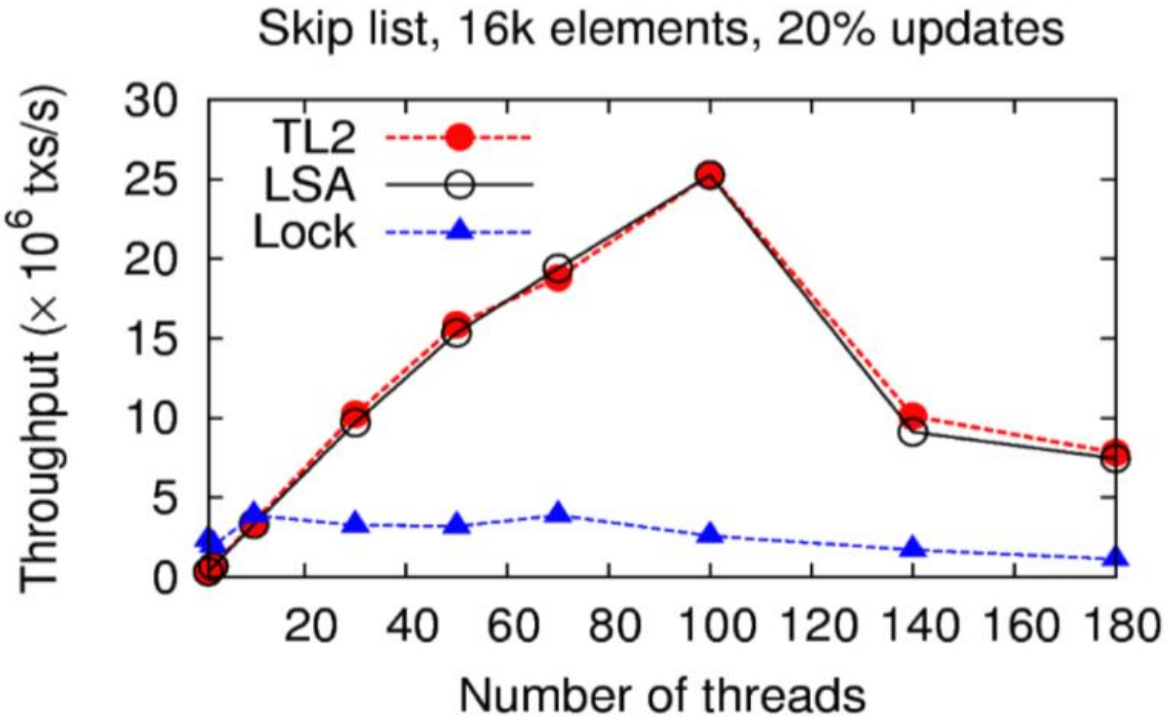
Transational Locking 2 [Dice2006]

- REMARKS:
  - Re-validating the read set before applying updates is required due to possible concurrent updates during write-set locking and GVC increment
- Read-only transactions
  - Do not need to increase GVC
  - Do not need to acquire any lock
  - Do not need to revalidate the read set
  - Do not need the read set

Transational Locking 2 [Dice2006]

- REMARKS:
  - Re-validating the read set before applying updates is required due to possible concurrent updates during write-set locking and GVC increment
- Read-only transactions
  - Do not need to increase GVC
  - Do not need to acquire any lock
  - Do not need to revalidate the read set
  - Do not need the read set

What about Software Transactional Memory

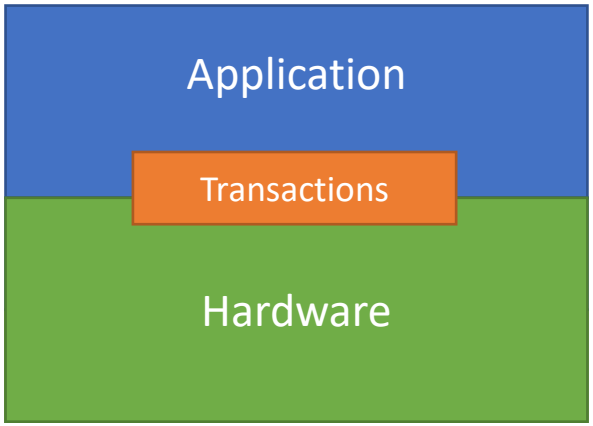


What about Software Transactional Memory

- Scale as (or better than) fine-grain locking
- Overheads hamper scalability
  - Due to instrumented access (overhead for each read/write)
  - Read set validation
- Hot topic in 2000s
  - A pletora of implementations for several programming languages
    - C/C++: TinySTM, G++ v4.7 (still expertimental)
    - C#: SXM by Microsoft (discontinued)
    - Haskell: STM is part of the Haskell platform
    - Scala: Akka framework
- Large debate on its practical impact
  - Software Transactional Memory: Why Is It Only a Research Toy?: The promise of STM may likely be undermined by its overheads and workload applicabilities. [Cascaval2008]
  - Transactional Memory Should Be an Implementation Technique, Not a Programming Interface [Boehm2009]
  - Why STM can be more than a Research Toy [Dragojević2011]

Hardware Transactional Memory

Intel TSX
BlueGene
RockProcessor
Arm Transactional Extension
IBM POWER8 and 9



Memory:

- Exploit cache coherency protocols
- Modified
- Exclusive
- Shared
- Invalid
- Tracked for speculative execution of transaction
- Losing track of a cache line leads to an abort

CPU:

- Ability to restore the processor state as the one before the beginning

Hardware transaction and abort

- Why can a hardware transaction abort?
  - Whenever, we lose track of a cache line….
- Any reason that could lead to an invalidation of a tracked cache line:
  - Another core wants it exclusive (conflict)
  - Change of execution mode (syscall, interrupts, page fault)
  - Working set too large
  - False cache sharing
- MESI:
  - https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/caches/MESI.htm
- TSX MESI:
  - https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/caches/TSX.htm

Intel Transactional Synchronization eXtensions (TSX)

RTE

- XBEGIN:
  - Start a hardware transaction (keep track of accessed cache lines)
- XEND:
  - Try to commit a hardware transaction (untrack cache lines)
- XABORT:
  - Make a hardware transaction abort programmatically

Are HTM so simple?

```
int committed_count;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
start_tsx:
    if(_XBEGIN() == _XBEGIN_STARTED){
        committed_count++;
        do_job(buf,...)
        _XEND();
        return;
    }
    else goto start_tsx;
}
```

All threads try to updated the same cache line!!!

Huge memory init!!!

This is not a good fallback path!

Are HTM so simple?

```
int committed_count;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
start_tsx:
    if(_XBEGIN() == _XBEGIN_STARTED){
        do_job(buf,...)
        _XEND();
        FAD(&committed_count,1);
        return;
    }
    else goto start_tsx;
}
```

Huge memory init!!!

This is not a good fallback path!

Are HTM so simple?

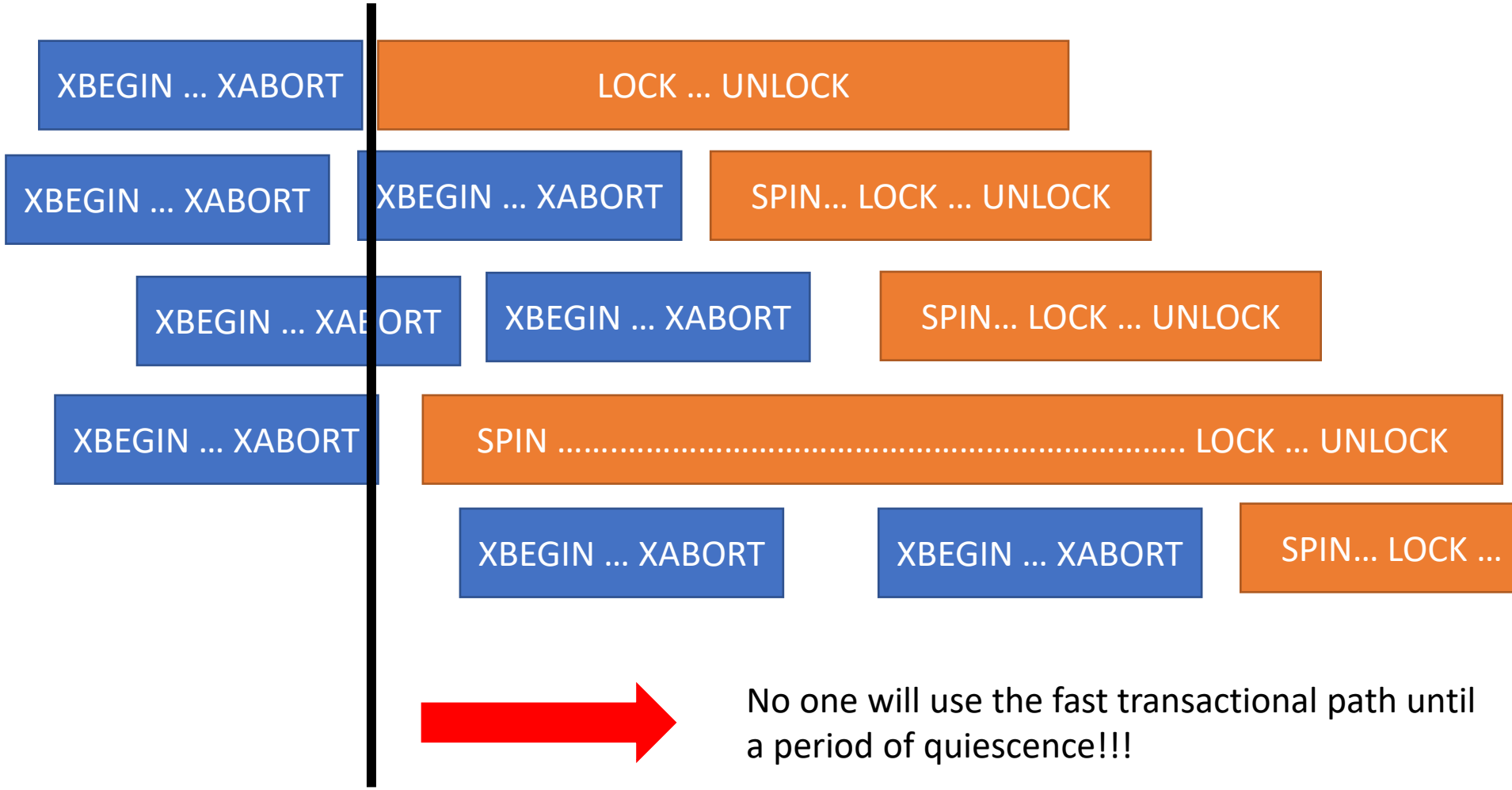
```
int committed_count;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
init(buf);
start_tsx:
      if(_XBEGIN() == _XBEGIN_STARTED){
            do_job(      ...)
            _XEND();
            FAD(&committed_count,1);
            return;
      }
      else goto start_tsx;
}
```

This is not a good fallback path!

Are HTM so simple? NO!

```
int committed_count; volatile int lock = UNLOCKED;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
init(buf); bool fb = false; int retry =0;
start_tsx:
     if(fb || _XBEGIN() == _XBEGIN_STARTED){
          if(lock==LOCKED) _XABORT();
          if(fb) TTAS(&lock, LOCKED);
          do_job(buf,...)
          if(fb) lock = UNLOCKED;
          _XEND();
          FAD(&committed_count,1);
          return;
     }
     else {fb=++retry>MAX_RETRY; goto start_tsx;}
}
```

Are NO! HTM so simple?



No one will use the fast transactional path until a period of quiescence!!!

Are HTM so simple? NO!

```
int committed_count; volatile int lock = UNLOCKED;
void transaction(){
char *buf = malloc(4096*1024); // 4MB
init(buf); bool fb = false; int retry =0;
start_tsx:
      if(fb || _XBEGIN() == _XBEGIN_STARTED){
            if(lock==LOCKED){while(lock==LOCKED);
                  _XABORT();}
            if(fb) TTAS(&lock, LOCKED);
            do_job(buf,...)
            if(fb) lock = UNLOCKED;
            _XEND();
            FAD(&committed_count,1);
            return;
      }
      else {fb=++retry<MAX_RETRY; goto start_tsx;}
}
```

Are NO! HTM so simple?

- We cannot replace lock with HTM as is due to performance aspects
- Naïve code might abort frequently due to:
  - Statistics
  - Memory allocations
  - Fallback path policy make the fast past rarely used
  - False cache-sharing
  - NUMA
  - NVRAM

Intel Transactional Synchronization eXtensions (TSX)

RTE

- XBEGIN:
  - Start a hardware transaction (keep track of accessed cache lines)
- XEND:
  - Try to commit a hardware transaction (untrack cache lines)
- XABORT:
  - Make a hardware transaction abort programmatically
- Needs a fallback path (e.g., by using locks)

HLE

- XACQUIRE:
  - Start a hardware transaction
  - execute a RMW without the LOCK prefix (XACQUIRE LOCK XCHG mutex, 1)
- XRELEASE:
  - Execute a mov to release the lock (XRELEASE mov mutex, 0)
  - Try to commit
- No need for an additional fallback path (just drop xacquire/xrelease and restart)

Is it worth investing in optimizing our code for HTM?

- VERY HARD TO SAY

HTM has been around for a while (2014), BUT:

- IBM BlueGene/Q — It is a high-end processor, not an off-the-shelf
- RockProcessor — Canceled in 2009
- IBM POWER8 and 9 (Power ISA v.2.07 to 3.0)
- Intel TSX — Not present in 10 (3.1)
  - First releases were bugged => disabled by firmware update
  - As other speculative components of Intel processors, they are vulnerable (leak info, see *TSX* Asynchronous Abort (TAA) / CVE-2019-11135) => disabled by firmware update
  - Not supported in Comet Lake and Ice Lake cpu (finger crossed for the next one)
  - Sapphire Rapids (2023) support TSX (and a new instruction TSXLDTRK), but still unsecure when Hyperthreading is enabled
- Arm Transactional Extension introduced in the last generation Armv9 (Mar 30 2021)

What about Software Transactional Memory

From a programmer perspective:

- It is less efficient than hardware implementation
- It generally provides stronger progress
- No need for a fallback path
- Processor independent
- Stick with the support of the community/organization developing it

What about Transactional Memory

- Topics we did not discuss about TM
  - Distributed STM
  - Heterogenous STM (CPU+GPU)
  - TM on NVRAM
  - Is opacity the actual reference correctness criteria?