CS636: Transactional Memory

Swarnendu Biswas

Semester 2018-2019-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Challenges with Concurrent Programming

Order, atomicity & sequential consistency violations

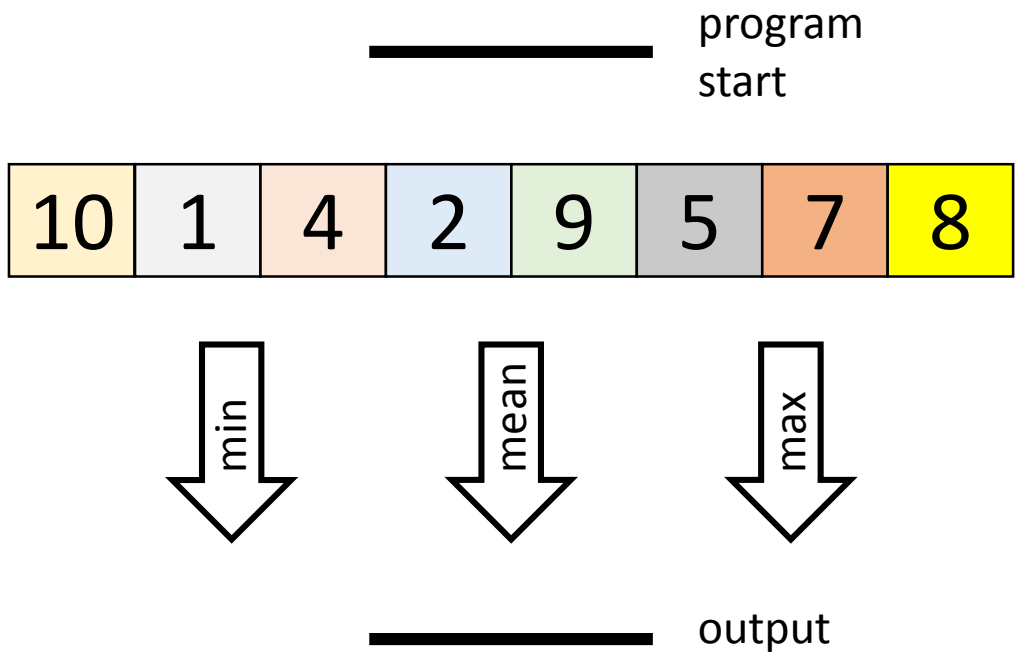
Concurrent and correct

Deadlock

Poor performance: lock contention, serialization

Task Parallelism

- Different tasks running on the same data
  - Threads execute computation concurrently
  - E.g., pipelines
- Explicit synchronization is used to coordinate threads



HashMap in Java

```
public Object get(Object key) {
  int idx = hash(key);          // Compute hash
  HashEntry e = buckets[idx];   // to find bucket
  while (e != null) {           // Find element in bucket
    if (equals(key, e.key))
      return e.value;
    e = e.next;
  }
  return null;
}
```

Synchronized HashMap in Java

```
public Object get(Object key) {
  synchronized (mutex) {  // mutex guards all accesses
    return myHashMap.get(key);
  }
}
```

- Uses explicit coarse-grained locking

Coarse-Grained and Fine-Grained Locking

Coarse-grained

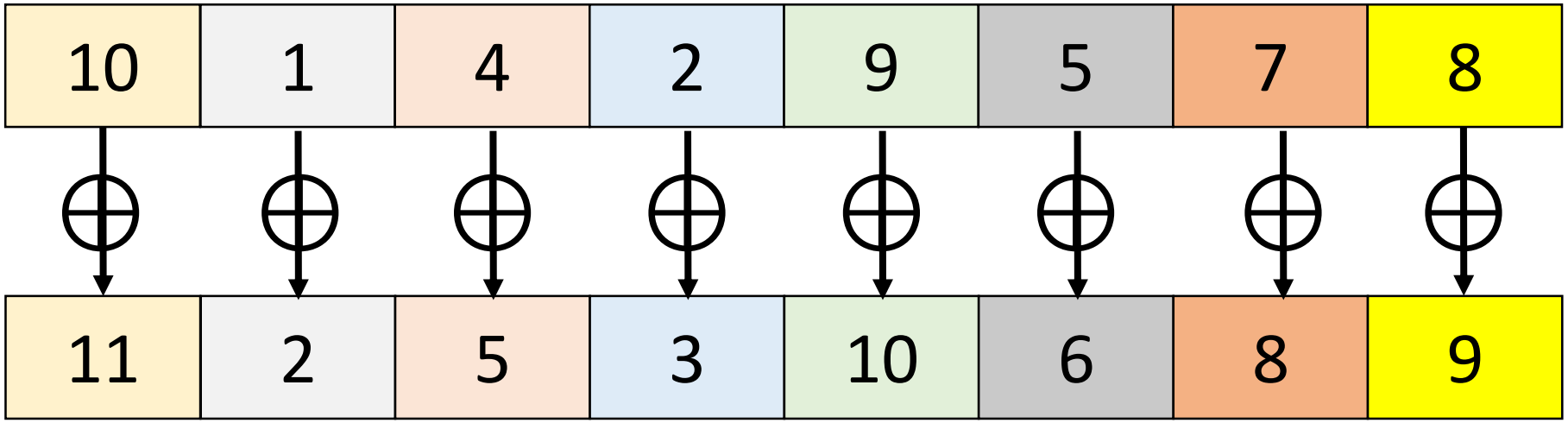
- **Pros**: Easy to implement
- **Cons**: limits concurrency, poor scalability

Fine-grained

- **Idea**: Use a separate lock per bucket
- **Pros**: thread safe, concurrent
- **Cons**: difficult to get correct, error-prone

Data Parallelism

- Same task applied on many data items in parallel
  - E.g., processing pixels in an image
- Useful for numeric computations
- Not an universal programming model



Task vs Data Parallelism

Task Parallelism	Data Parallelism
• Different operations on same or different data • Parallelization depends on task decomposition	• Same operation on different data • Parallelization proportional to the input data size

Combining Task and Data Parallelism

Processing in graphics processors

Task parallelism through pipelining

- Each task could apply a filter in a series of filters

Data parallelism for a given filter

- Apply the filter computation in parallel for all pixels

https://www.zdnet.com/article/understanding-task-and-data-parallelism-3039289129/

Abstraction and Composability

Abstraction

- Simplified view of an entity or a problem
- Example: procedures, ADT

Composability

- Join smaller units to form larger, more complex unit
- Example: library methods

Locks are difficult to program!

- If a thread holding a lock is delayed, other contending threads cannot make progress
  - All contending threads will possibly wake up, but only one can make progress
- Lost wakeups – missed notify for condition variable
- Deadlocks
- Priority inversion
- Locking relies on programmer conventions

Locking relies on programmer conventions!

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

Actual comment from Linux Kernel

```
/*
 * 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.
 */
```

Bradley Kuszmaul, and Maurice Herlihy and Nir Shavit

Lock-based Synchronization is not Composable

```
class HashTable {
  void synchronized insert(T elem);
  boolean synchronized remove(T elem);
}
```

Say now you want to add a new method:

```
boolean move(HashTable tab1, HashTable tab2, T elem)
```

Choosing the right locks!

- Locking schemes for 4 threads may not be the most efficient at 64 threads
  - Need to profile the amount of contention

What about hardware atomic primitives?

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 Transactional Memory (TM)

- Provides **reasonable tradeoff** between abstraction and performance
  - No need for explicit locking
    - Avoids lock-related issues like lock convoying, priority inversion, and deadlocks

```
boolean move(HashTable tab1, HashTable tab2, T elem) {
  atomic {
    boolean res = tab1.remove(elem);
    if (res)
      tab2.insert(elem);
  }
  return res;
}
```

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

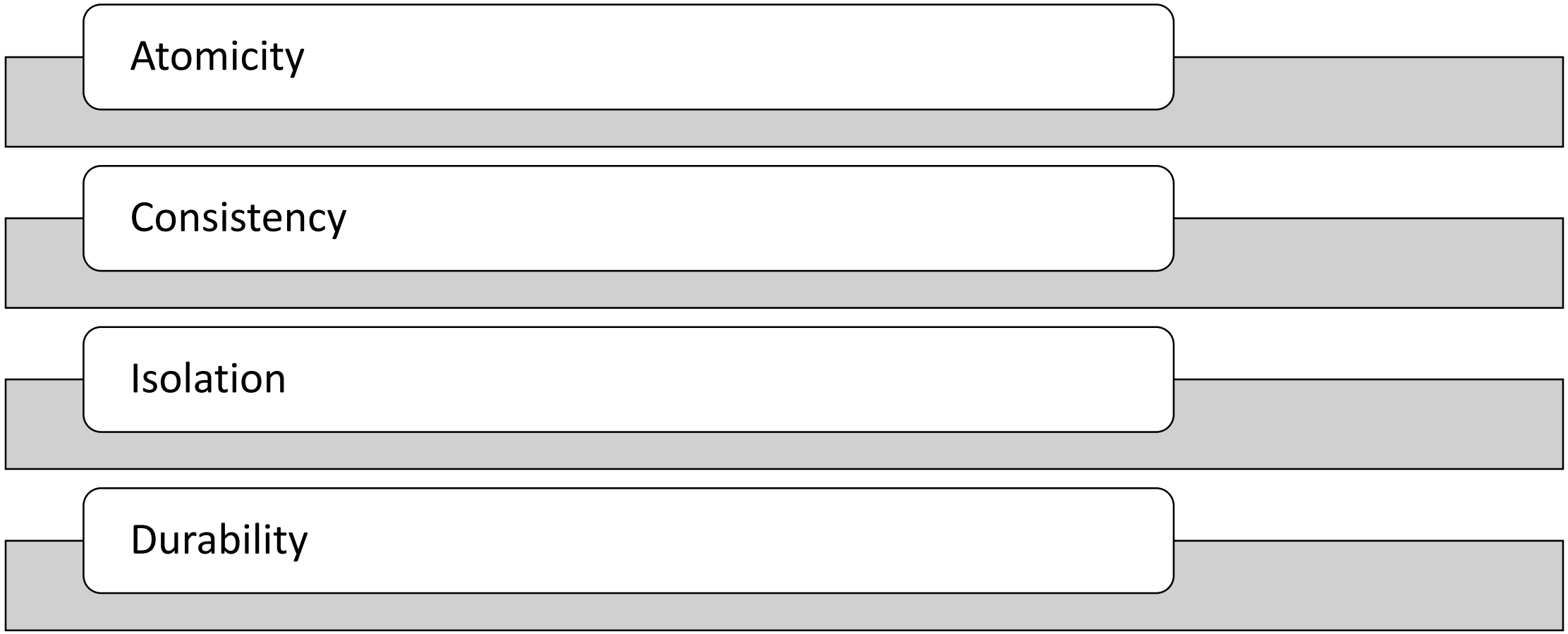
Basic TM Design

- 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**

Database Systems as a Motivation

- Database systems have successfully exploited parallel hardware for decades
- Achieve good performance by executing many queries simultaneously and by running queries on multiple processors when possible

Database Systems as a Motivation



TM vs Database Transactions

Database Transactions	TM
• Application level concept	• Supported by language runtime or hardware
• Durable	• Not durable
• Operations involve mostly disk accesses	• Operations are from main memory, performance is critical

Properties of TM execution

Tx		
	Atomic	appears to happen instantaneously
	Commit	Appears atomic
	Abort	Has no side effects
	Serializable	appear to happen serially in order
	Isolation	Other code cannot observe writers before commit

TM Execution Semantics

Thread 1

```
atomic {
  a = a – 20;
  b = b + 20;
  c = a + b;
  a = a – b;
}
```

Thread 1's updates to a, b, and c are atomic

Thread 2

```
atomic {
  c = c + 40;
  d = a + b + c;
}
```

Thread 2's either sees ALL updates to a, b, and c from T1 or NONE

No data race due to TM semantics

Atomicity violation

```
if (thd->proc_info)

  fputs(thd->proc_info, …)
```

time

```
…

thd->proc_info = NULL;
…
```

MySQL
ha_innodb.cc

Fixing Atomicity Violations with TM

```
atomic {
  if (thd->proc_info)
    fputs(thd->proc_info, …)
}
```

time

No data race due to TM semantics

```
atomic {
  thd->proc_info = NULL;
}
```

Fixing Atomicity Violations with TM

```
atomic {
  thd->proc_info = NULL;
}
```

time

```
atomic {
  if (thd->proc_info)
    fputs(thd->proc_info, …)
}
```

No data race due to TM semantics

Transactional HashMap

Pros	Cons
• Thread-safe, easy to program	• Good performance and scalability depends on the TM implementation

synchronized in Java

synchronized

- Provides mutual exclusion compared to other blocks on the same lock
- Nested blocks can deadlock if locks are acquired in wrong order

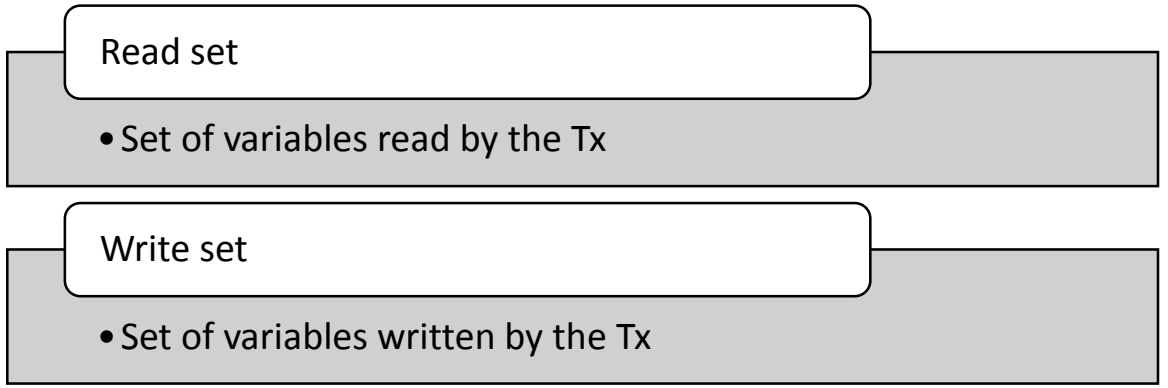
TM transaction

- A transaction is atomic w.r.t. all other transactions in the system
- Nested transactions never deadlock

TM Interface

```
void startTx();
bool commitTx();
void abortTx();
```

```
T readTx(T *addr);
void writeTx(T *addr, T val);
```



Transactions cannot replace all uses of locks!

Thread 1

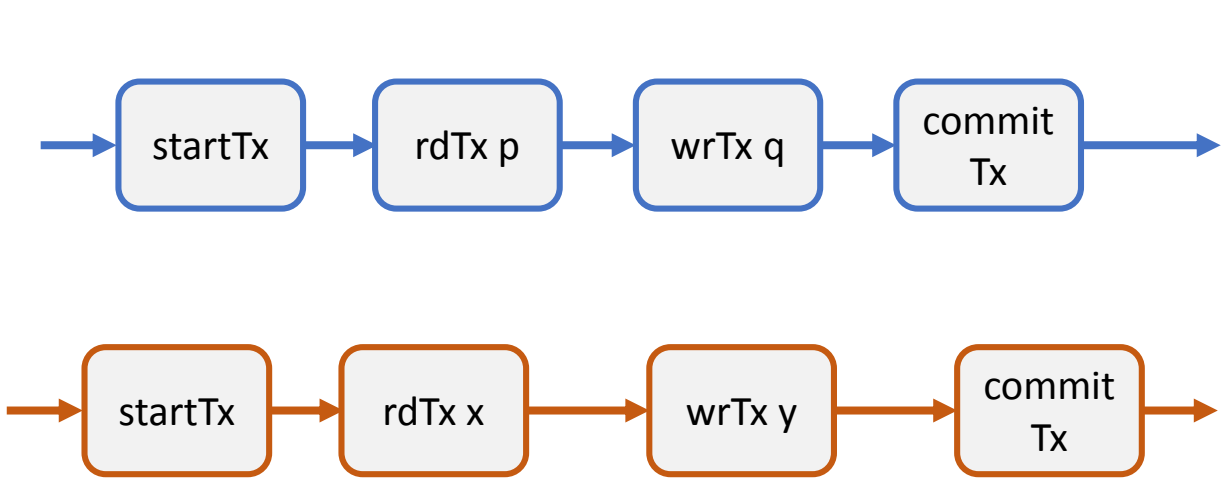
```
do {
  startTx();
  writeTx(&x, 1);
} while (!commitTx());
```

Thread 2

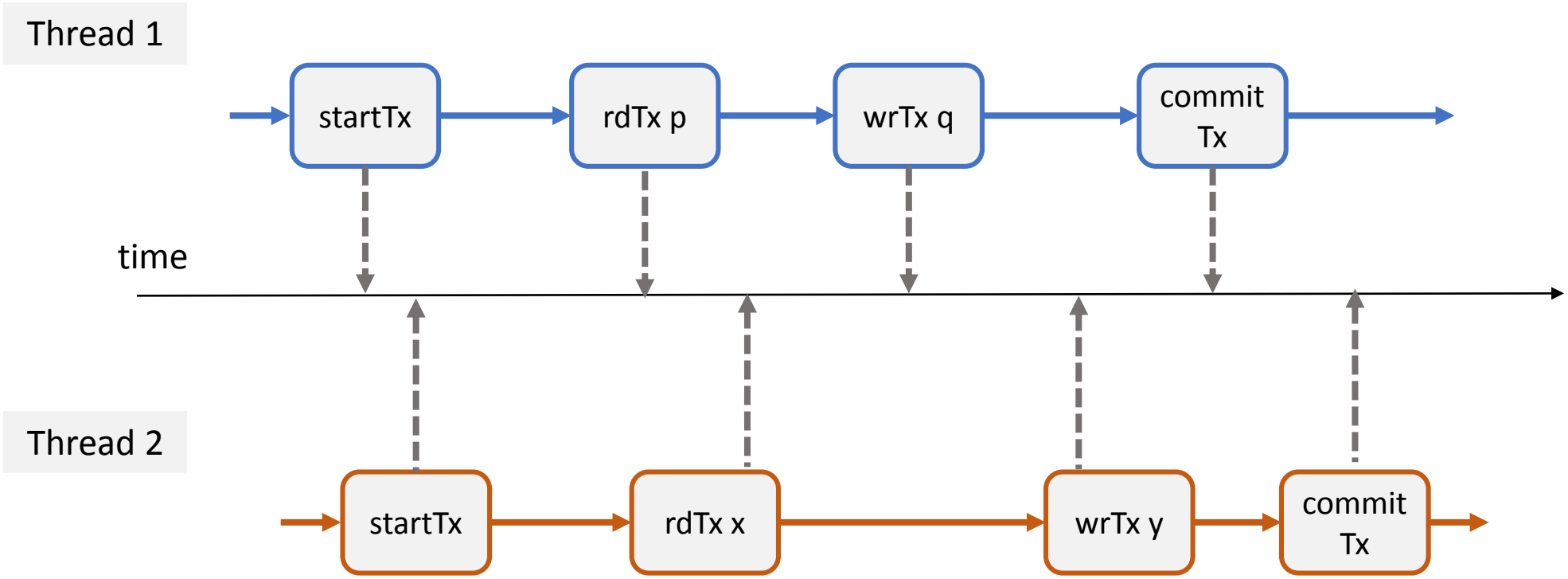
```
do {
  startTx();
  int tmp = readTx(&x);
  while (tmp == 0) {}
} while (!commitTx());
```

Concurrency in TM

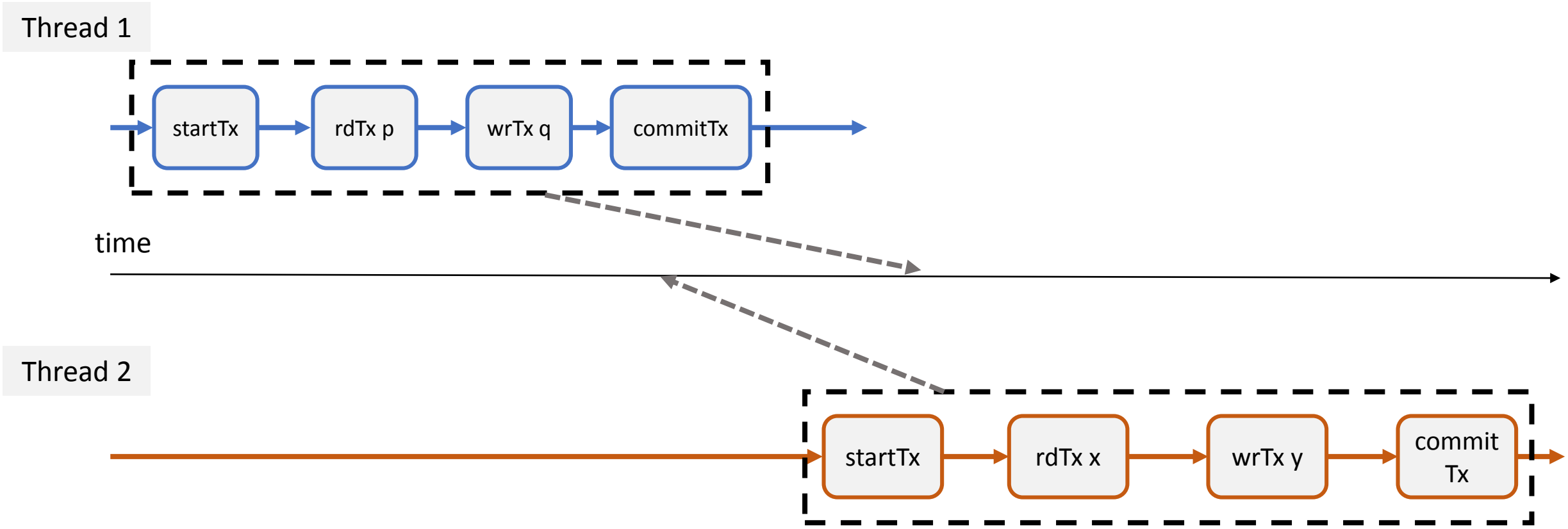
- Two levels
  - Among Txs
  - Among individual Tx operations



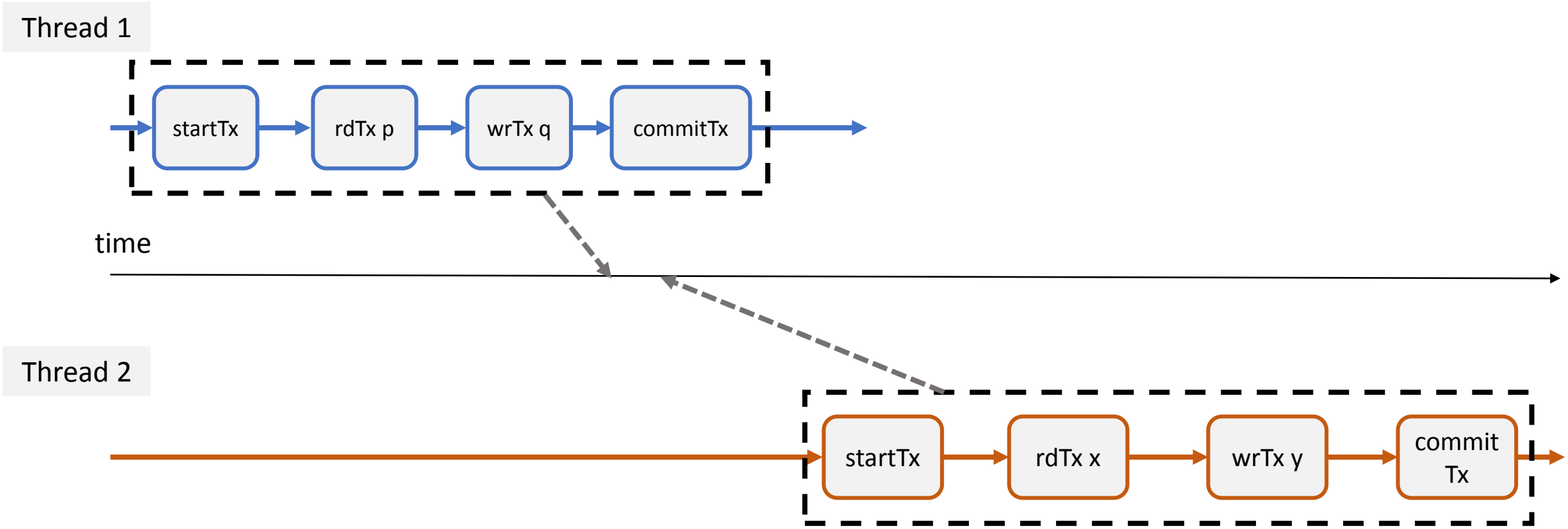
Linearizability



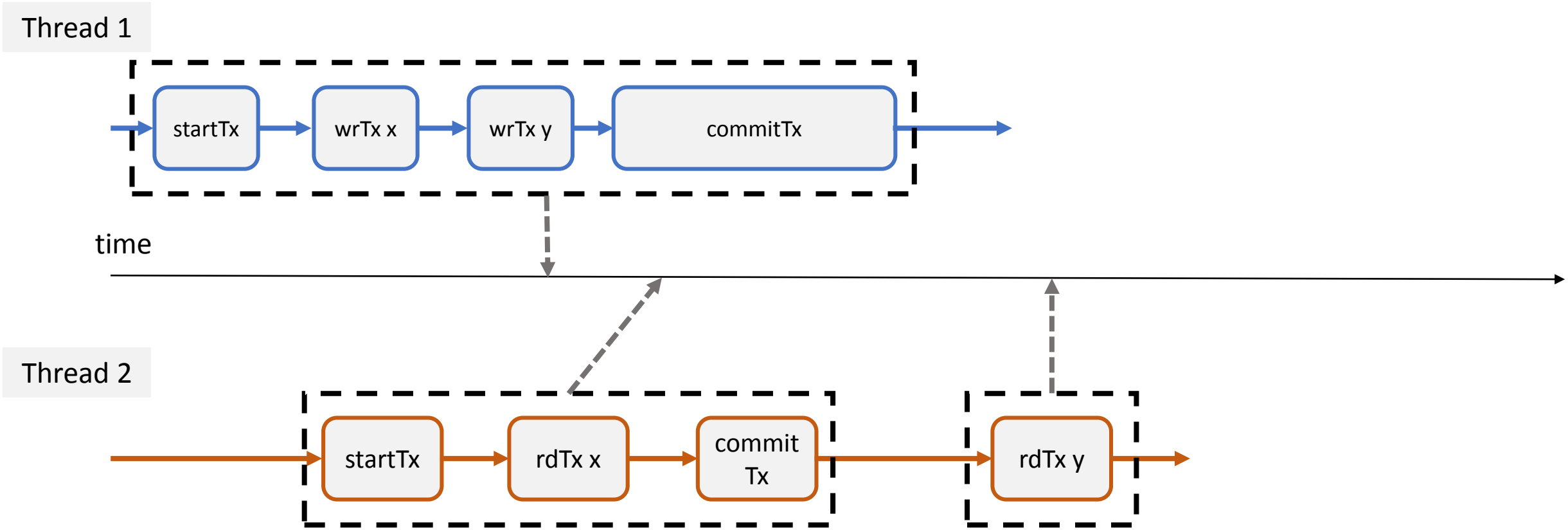
Serializability



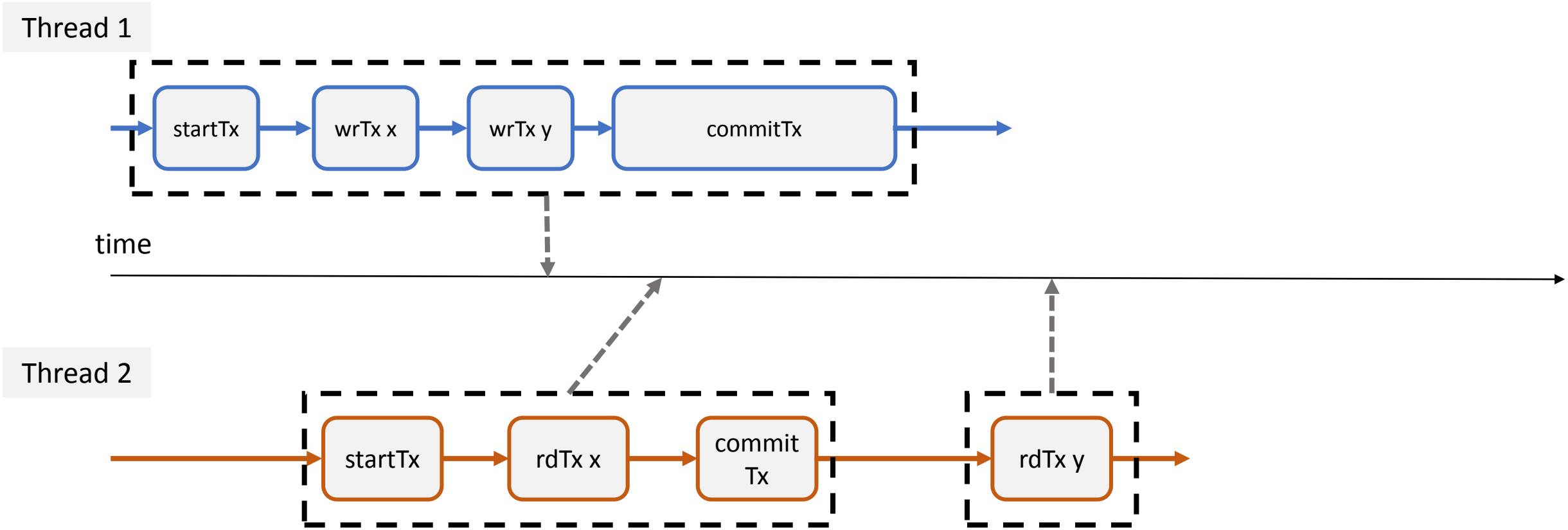
Strict Serializability



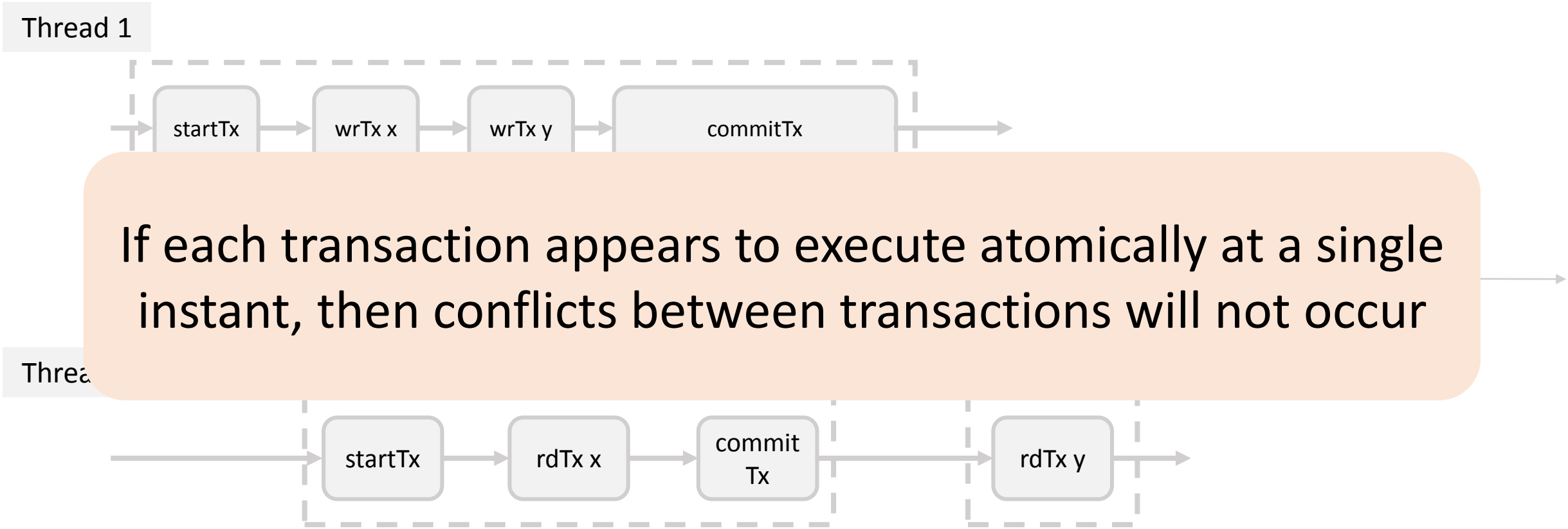
Limitations of Strict Serializability



Can Linearizability help with this?



Can Linearizability help with this?



Snapshot Isolation (SI)

Weaker isolation requirement than serializability

- Can potentially allow greater concurrency between Txs
- Many database implementations actually provide SI

SI allows a Tx's reads to be serialized before the Tx's writes

All reads must see a valid snapshot of memory

Updates must not conflict

Example of SI

```
x = 0
y = 0
```

Thread 1

```
do {
  startTx();
  int tmp_x = readTx(x);
  int tmp_y = readTx(y);
  int tmp = tmp_x + tmp_y + 1;
  writeTx(x, tmp);
} while (!commitTx());
```

Thread 2

```
do {
  startTx();
  int tmp_x = readTx(x);
  int tmp_y = readTx(y);
  int tmp = tmp_x + tmp_y + 1;
  writeTx(y, tmp);
} while (!commitTx());
```

Understanding SI

Data races are there for a purpose!

```
x = 0
y = 0
```

Sequentially consistent but not SI

```
int t = x + 1; (1)
x = t;
```

```
int t = x + 1; (1)
x = t;
```

SI but not sequentially consistent and not serializable

```
x = 1;
int t = y; (0)
```

```
y = 1;
int t = x; (0)
```

M. Zhang et al. Avoiding Consistency ExceptionsUnder Strong Memory Models. ISMM 2017.

TM Terminology

A **conflict occurs** when two transactions perform conflicting operations on the same memory location.

Let R_i and W_i be the read and write sets of Tx i. Then a conflict occurs iff

- $R_i \cap Wj \neq \emptyset$, or
- $W_i \cap Wj \neq \emptyset$, or
- $W_i \cap Rj \neq \emptyset$

TM Terminology

The **conflict is detected** when the underlying TM system determines that the conflict has occurred.

The **conflict is resolved** when the underlying TM system takes some action to avoid the conflict.

TM: Example Execution

bal = 1000

```
atomic {
  tmp = bal;
  bal = tmp + 100;
}
```

Location	Value read	Value written

```
atomic {
  tmp = bal;
  bal = tmp - 100;
}
```

Location	Value read	Value written

TM: Example Execution

bal = 1000

```
atomic {
  tmp = bal;
  bal = tmp + 100;
}
```

1

```
atomic {
  tmp = bal;
  bal = tmp - 100;
}
```

Location	Value read	Value written

Location	Value read	Value written
bal	1000	

TM: Example Execution

bal = 1000

```
atomic {
  tmp = bal;
  bal = tmp + 100;
}
```

2

```
atomic {
  tmp = bal;
  bal = tmp - 100;
}
```

Location	Value read	Value written
bal	1000	

Location	Value read	Value written
bal	1000	

TM: Example Execution

bal = 1000

```
atomic {
   tmp = bal;
   bal = tmp + 100;
}
```

3

```
atomic {
   tmp = bal;
   bal = tmp - 100;
}
```

Location	Value read	Value written
bal	1000	1100

Location	Value read	Value written
bal	1000	

TM: Example Execution

bal = 1100

```
atomic {
   tmp = bal;
   bal = tmp + 100;
}
```

3

```
atomic {
   tmp = bal;
   bal = tmp - 100;
```

Thread 1's Tx ends, **updates are committed**, value of `bal` is written to memory; Tx log is discarded

Location	Value read	Value written
bal	1000	**1100**

	Value read	Value written
bal	1000	

TM: Example Execution

bal = 1100

```
atomic {
   tmp = bal;
   bal = tmp + 100;
}
```

4

```
atomic {
   tmp = bal;
   bal = tmp - 100;
}
```

Location	Value read	Value written
bal	1000	900

TM: Example Execution

bal = 1100

```
atomic {
   tmp = bal;
   bal = tmp + 100;
}
```

4

```
atomic {
   tmp = bal;
   bal = tmp - 100;
}
```

Thread 2's Tx ends, but Tx **commit fails**, because value of `bal` in memory does not match the read log; Tx needs to rerun

Location	Value read	Value written
bal	1000	900

Concurrency Control

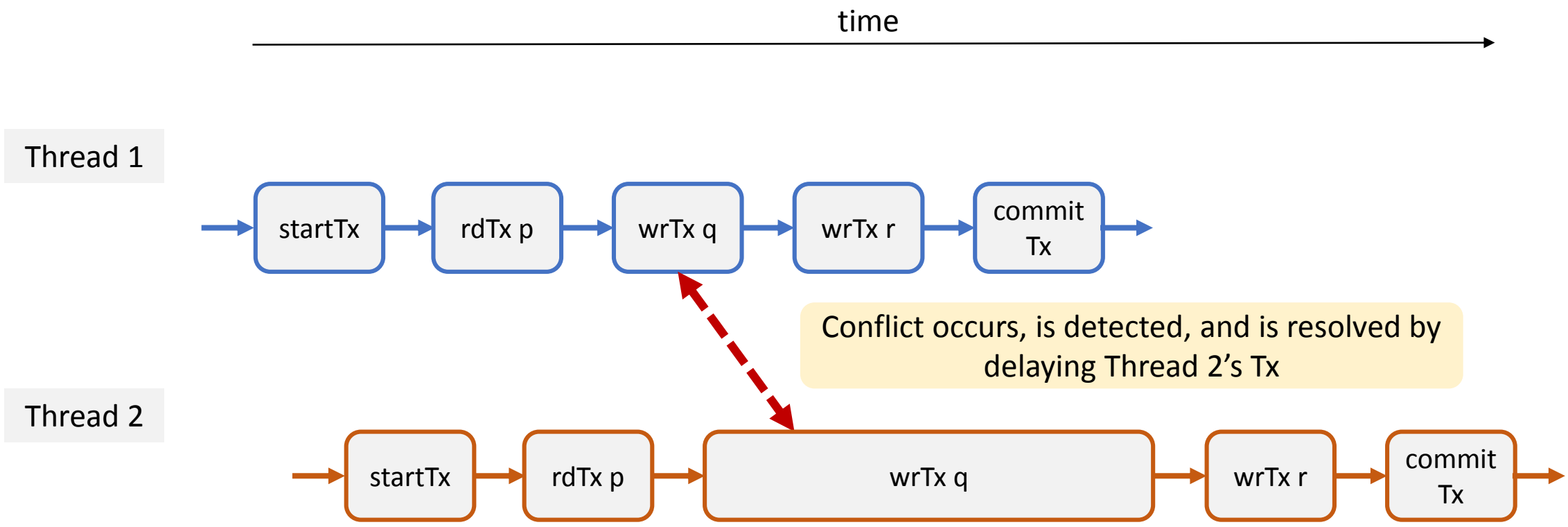
Pessimistic

- Occurrence, detection, and resolution happen **at the same time** during execution

Optimistic

- Conflict detection and resolution can happen **after** the conflict occurs

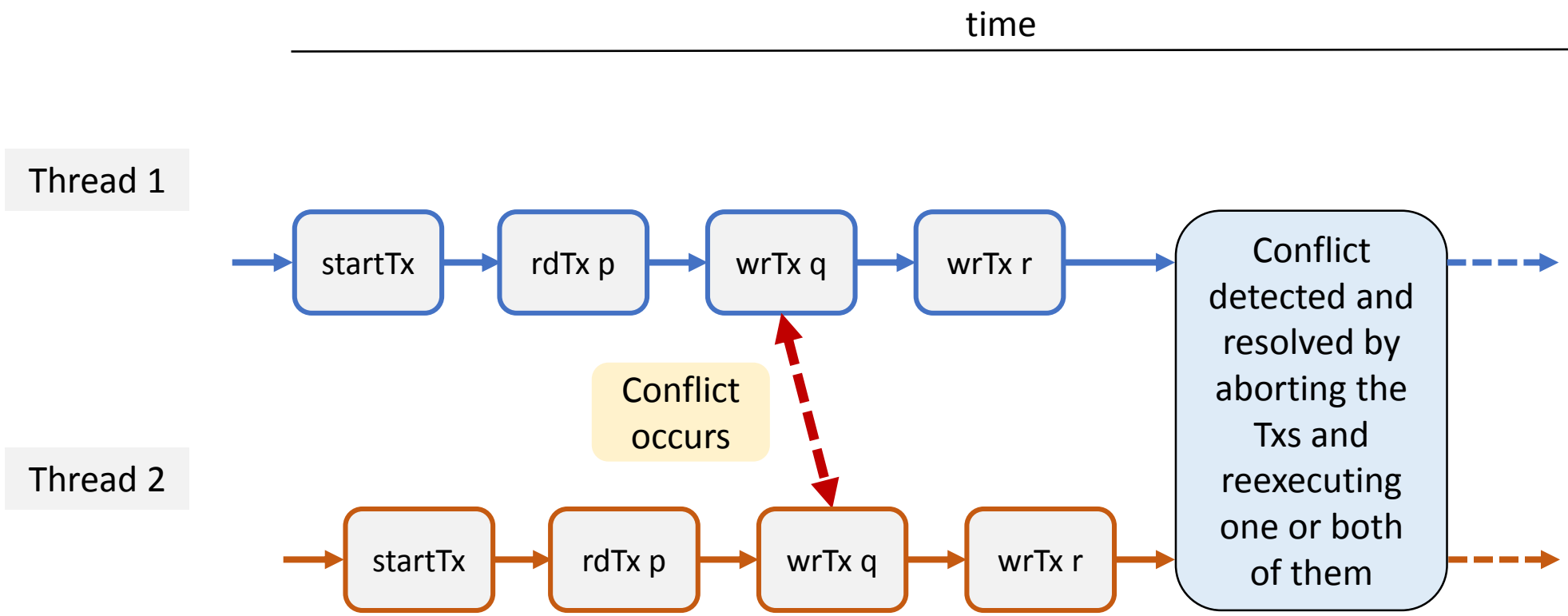
Pessimistic Concurrency Control



Time of locking

- When the Tx first accesses a location
- When the Tx is about to commit

Optimistic Concurrency Control



Concurrency Control

Pessimistic	Optimistic
• Usually claims exclusive ownership of data before accessing • Effective in high contention cases • Needs to avoid deadlock situations	• Avoids claiming exclusive ownership of data • Effective in low contention cases • Needs to avoid livelock situations

Hybrid Concurrency Control

Use pessimistic control for writes and optimistic control for reads

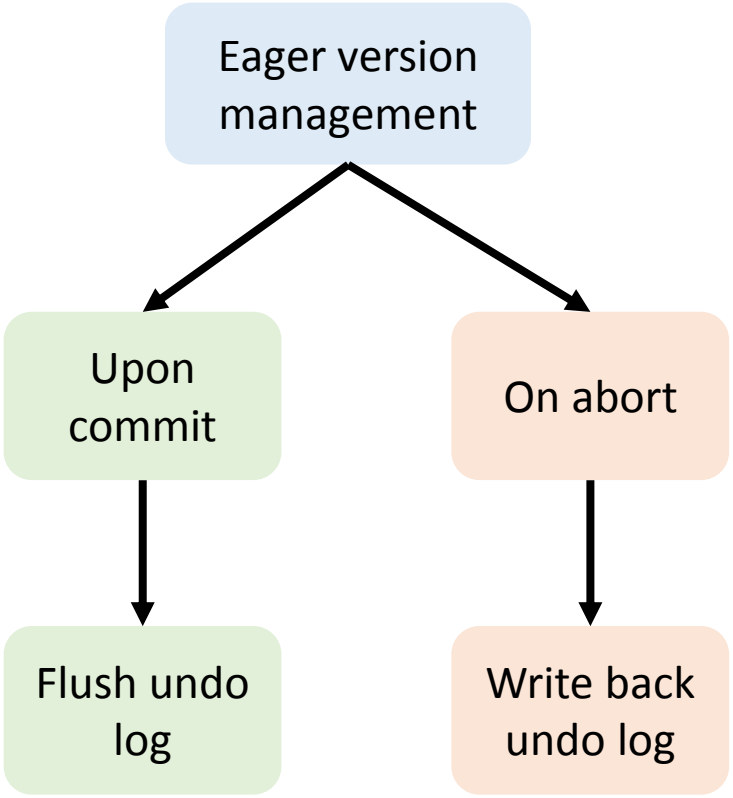
Use optimistic control TM with pessimistic control of irrevocable Txs

Version Management

TMs need to track updates for conflict resolution

Eager

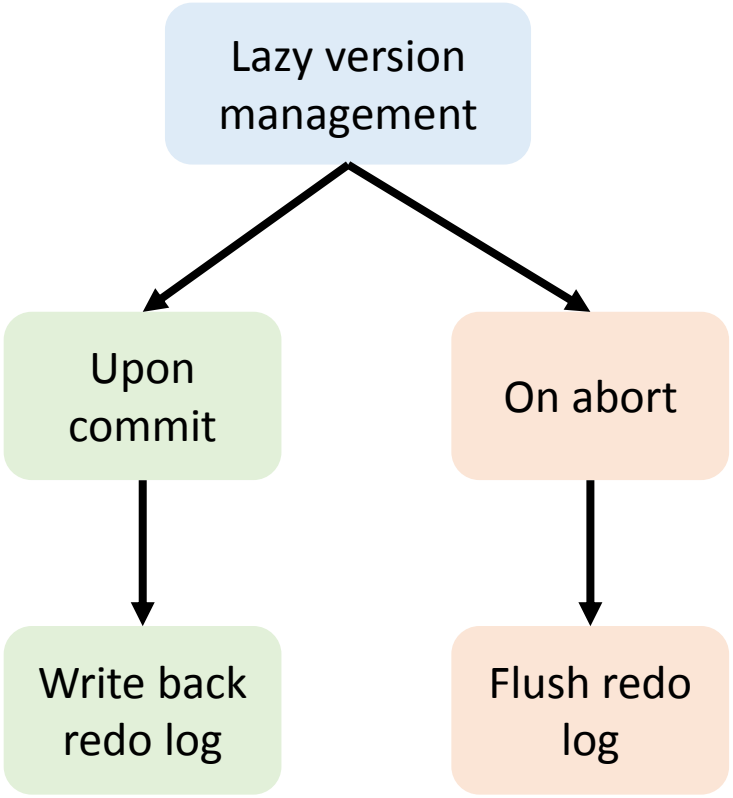
- Tx directly updates data in memory (direct update)
- Maintains an undo log with overwritten values
- Values in the undo log are used to revert updates on an abort



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



Conflict Detection

Pessimistic concurrency control is straightforward

How do you check for conflicts in optimistic concurrency control?

Conflict Detection

Pessimistic concurrency control is straightforward

How do you check for conflicts in optimistic concurrency control?

- Validation operation – Successful validation means Tx had no conflicts

Conflict Detection in Optimistic Concurrency Control

Conflict granularity

- Object or field, line offset or cache line

Time of conflict detection

- Just before access (eager), during validation, during commit (lazy)

Conflicting access types

- Among concurrent Txs, or between active and committed Txs

Object Layout

HEADER
field1
field2
field3

Object layout

Object Model in Jikes RVM

```
|<- lo memory                                                    hi memory ->|

  SCALAR LAYOUT:
|<---------- scalar header --------->|
+---------+-----------+-----------+------+------+------+--------+
| GCHeader | MiscHeader | JavaHeader | fld0 | fld1 | fldx | fldN-1 |
+---------+-----------+-----------+------+------+------+--------+
                       ^ JHOFF              ^objref
```

https://www.jikesrvm.org/JavaDoc/org/jikesrvm/objectmodel/ObjectModel.html

Issues with Conflict Granularity

```
x = 0
y = 0
```

Thread 1

```
do {
  startTx();
  tmp = readTx(&x);
  writeTx(x, 10);
} while (!commitTx());
```

Thread 2

```
…
y = 20;
…
```

Question

- Eager version management
  - Should you use pessimistic or optimistic control?

Inconsistent Reads and Zombie Txs

```
x = 0
y = 0
```

Thread 1

```
do {
  startTx();
  int tmp1 = readTx(&x);

  int tmp2 = readTx(&y);
  while (tmp1 != tmp2) {}
} while (!commitTx());
```

Thread 2

```
do {
  startTx();
  writeTx(&x, 10);
  writeTx(&y, 10);
} while (!commitTx());
```

Weak and Strong Atomicity

Weak

- Provides Tx semantics only among Txs
- Checks for conflicts only among Txs

Strong

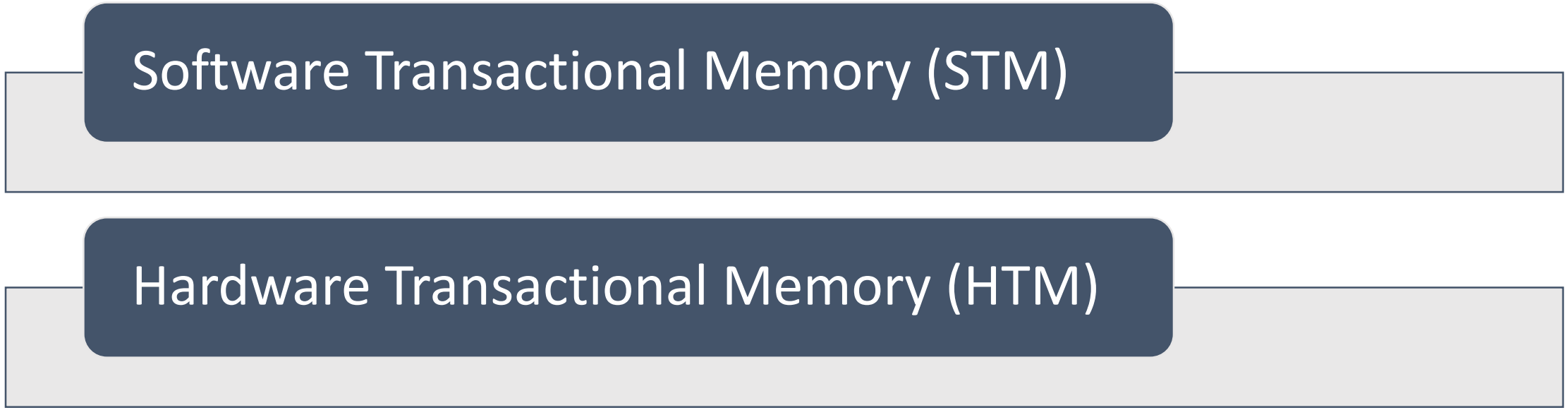
- Guarantees Tx semantics among Txs and non-Txs

Often referred to as weak and strong isolation

Few Issues to Consider with Weak Isolation

- Non-repeatable reads
- Intermediate lost updates
- Intermediate dirty reads
- Granular lost updates
- …
- …

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

Software Transactional Memory

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

We love questions!

Well-designed applications should have low conflict rates

Is design of undo log important in eager version management?

Is design of redo log important in lazy version management?

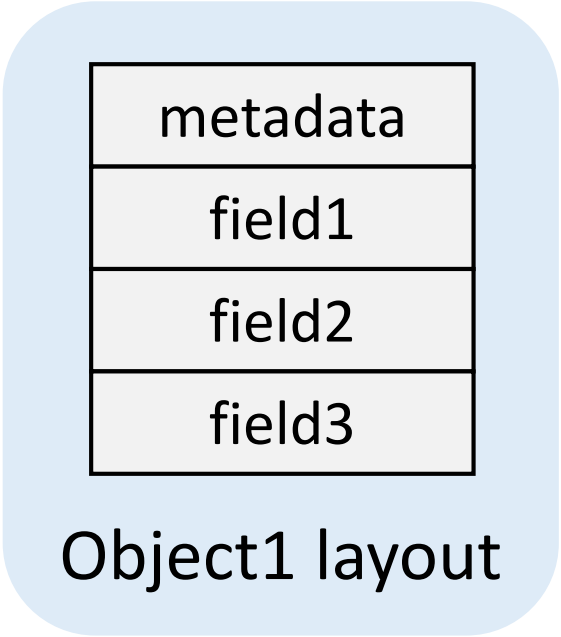
Implementing STM

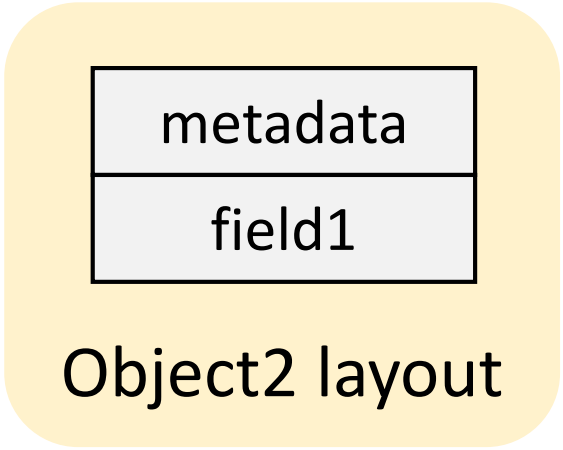
- Use compilation passes to instrument the program
  - startTx() - Tx entry point (prolog)
  - commitTx() - exit point (epilog)
  - readTx/writeTx - Transactional read/write accesses
- TM runtime tracks memory accesses, detects conflicts, and commits/aborts Txs

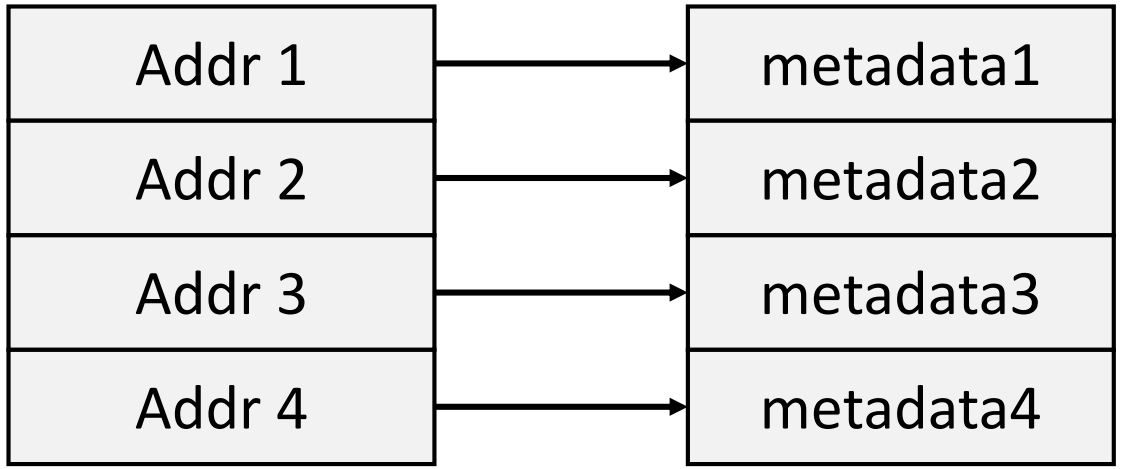
```
atomic {
   tmp = x;
   y = tmp + 1;
}
```

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

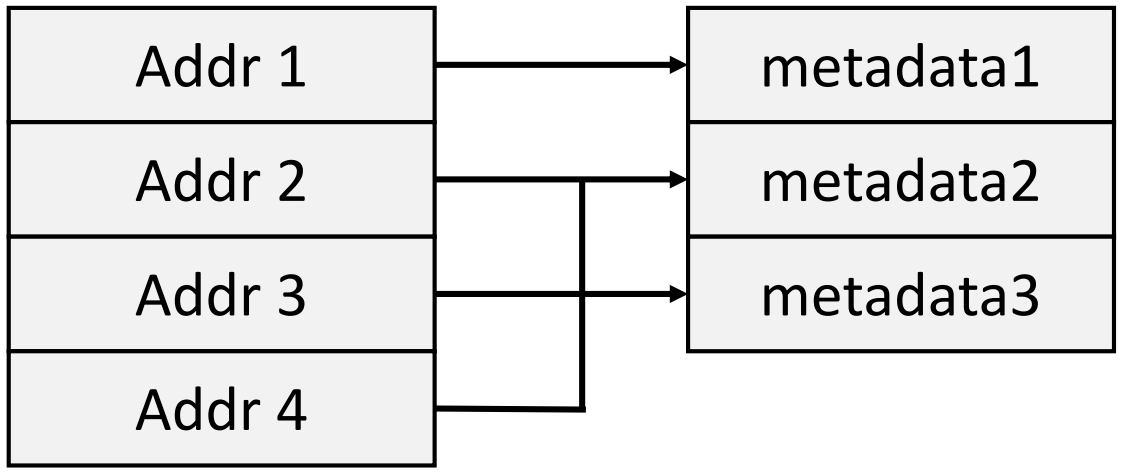
Object Metadata and Word Metadata

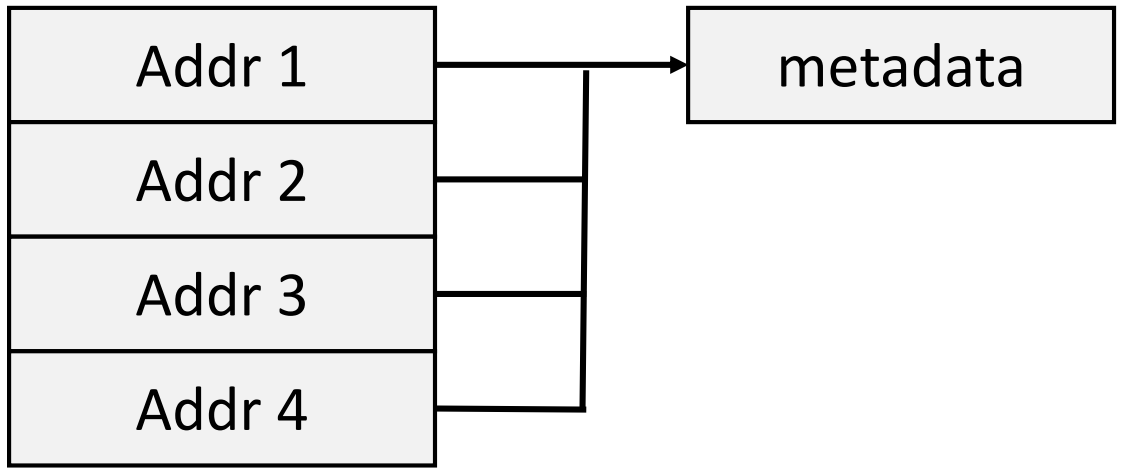






Variants of Word-based Metadata

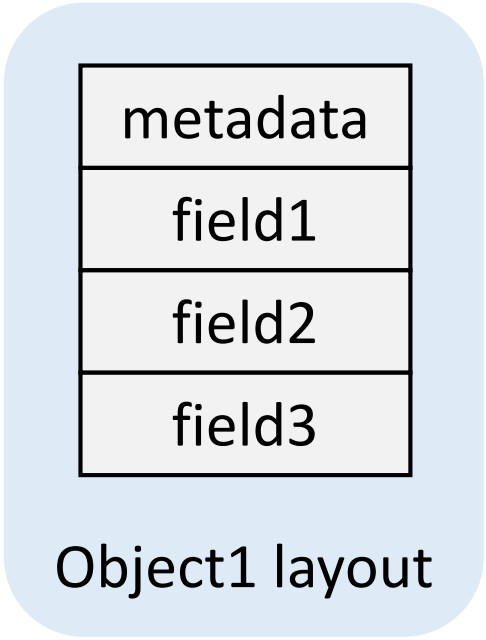


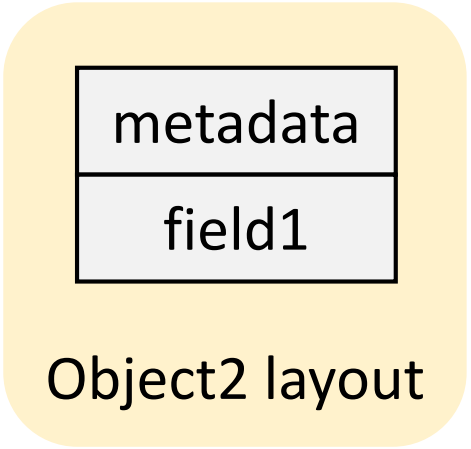


Use hash functions to map addresses to a fixed-size metadata space

Process-wide metadata space

Pros and Cons of Object Metadata





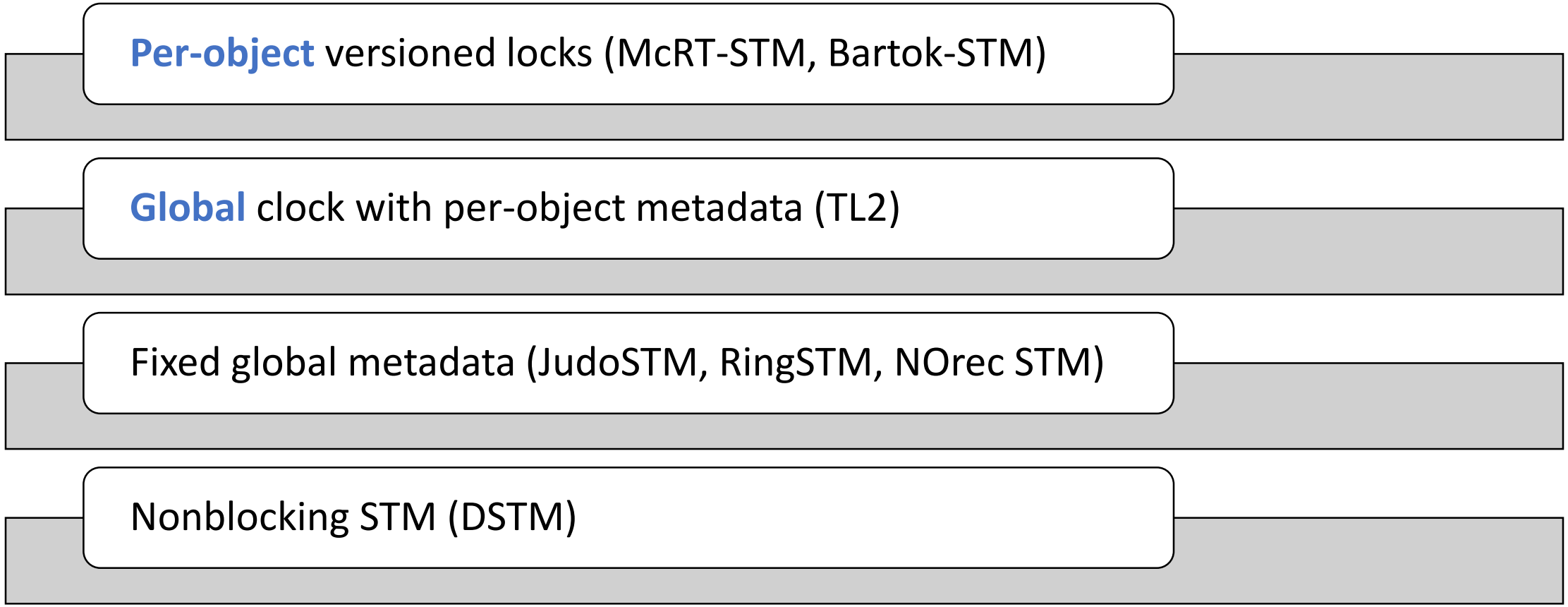
Pros

- Metadata lies on the same cache line
- Single update for accesses to all fields

Cons

- Potential for false conflicts
- Increases coupling
  - E.g., GC considerations

Major STM Designs



Which granularity to use?

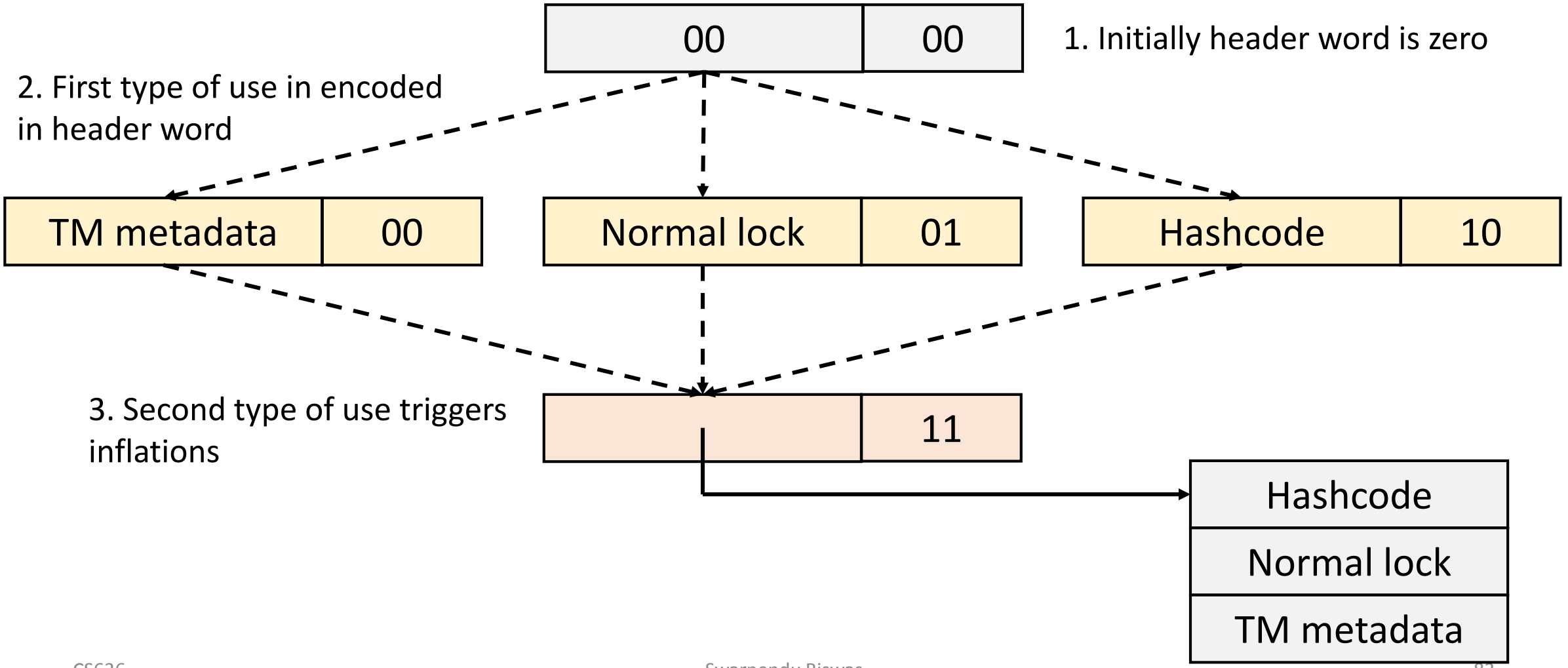
Impact on memory usage

Impact on performance

- Speed of mapping location to metadata

Potential impact due to false conflicts

Header Word Optimizations



Lock-Based STM with Versioned Reads

High-level design	Pessimistic concurrency-control for writes	Locks are acquired dynamically
	Optimistic concurrency control for reads	Validation using version numbers

Other Design Issues

- Eager vs lazy version management
- Access-time locking or commit-time locking

STM Metadata

Versioned locks

- Lock – To arbitrate writes
- Version number – detect conflicts involving reads

- Lock is available – no pending writes, holds the current version of the object
- Lock is taken – refers to the owner Tx

Read and Write Operations

```
readTx(tx, obj, off) {
  tx.readSet.obj = obj;
  tx.readSet.ver = getVerFromMetadata(obj);
  tx.readSet++;

  return read(obj, off);
}
```

```
writeTx(tx, obj, off, newVal) {
  acquire(obj);

  tx.undoLog.obj = obj;
  tx.undoLog.offset = off,
  tx.undoLog.val = read(obj, off);
  tx.undoLog++;

  tx.writeSet.obj = obj;
  tx.writeSet.off = off;
  tx.writeSet.ver = ver;
  tx.writeSet++;

  write(obj, off, newVal);
  release(obj);
}
```

Eager version management

Read and Write Operations

```
readTx(tx, obj, off) {
  tx.readSet.obj = obj;
  tx.readSet.ver = getVerFromMetadata(obj);
  tx.readSet++;

  return read(obj, off);
}
```

Eager version management

```
writeTx(tx, obj, off, newVal) {
  acquire(obj);
  undoLogInt(tx, obj, off);
  tx.writeSet.obj = obj;
  tx.writeSet.off = off;
  tx.writeSet.ver = ver;
  tx.writeSet++;
  write(obj, off, newVal);
  release(obj);
}

undoLogInt(tx, obj, off) {
  tx.undoLog.obj = obj;
  tx.undoLog.offset = off,
  tx.undoLog.val = read(obj, off);
  tx.undoLog++;
}
```

Conflict Detection on Writes

Writes? **Reads**

Conflict Detection on Reads

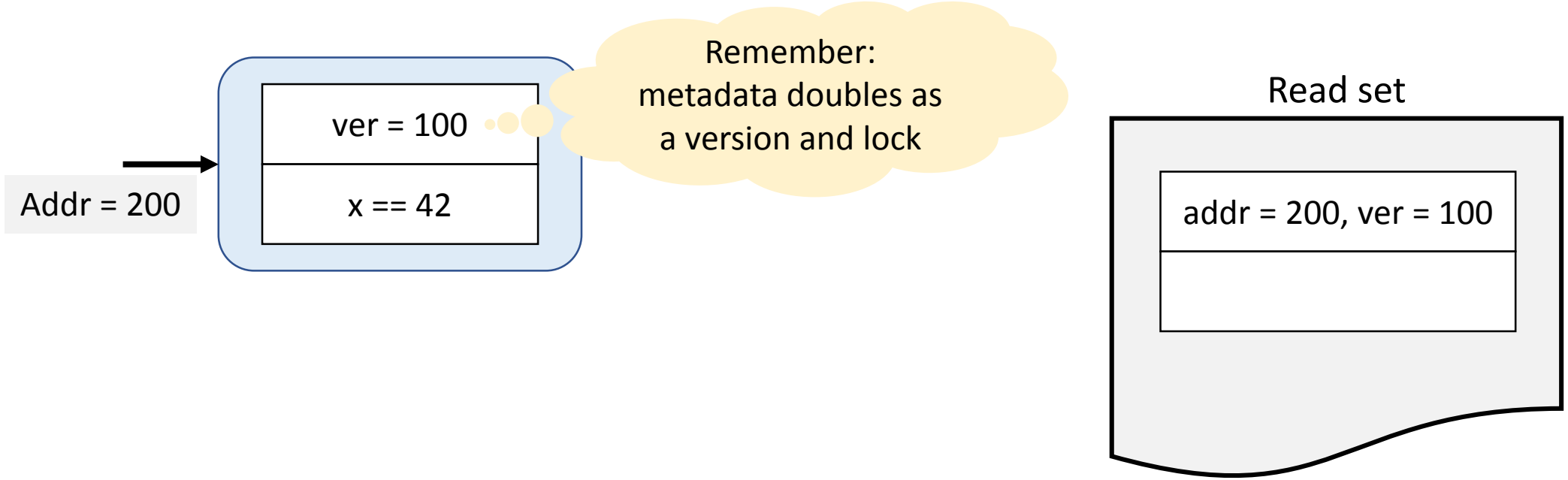
Writes

Reads?

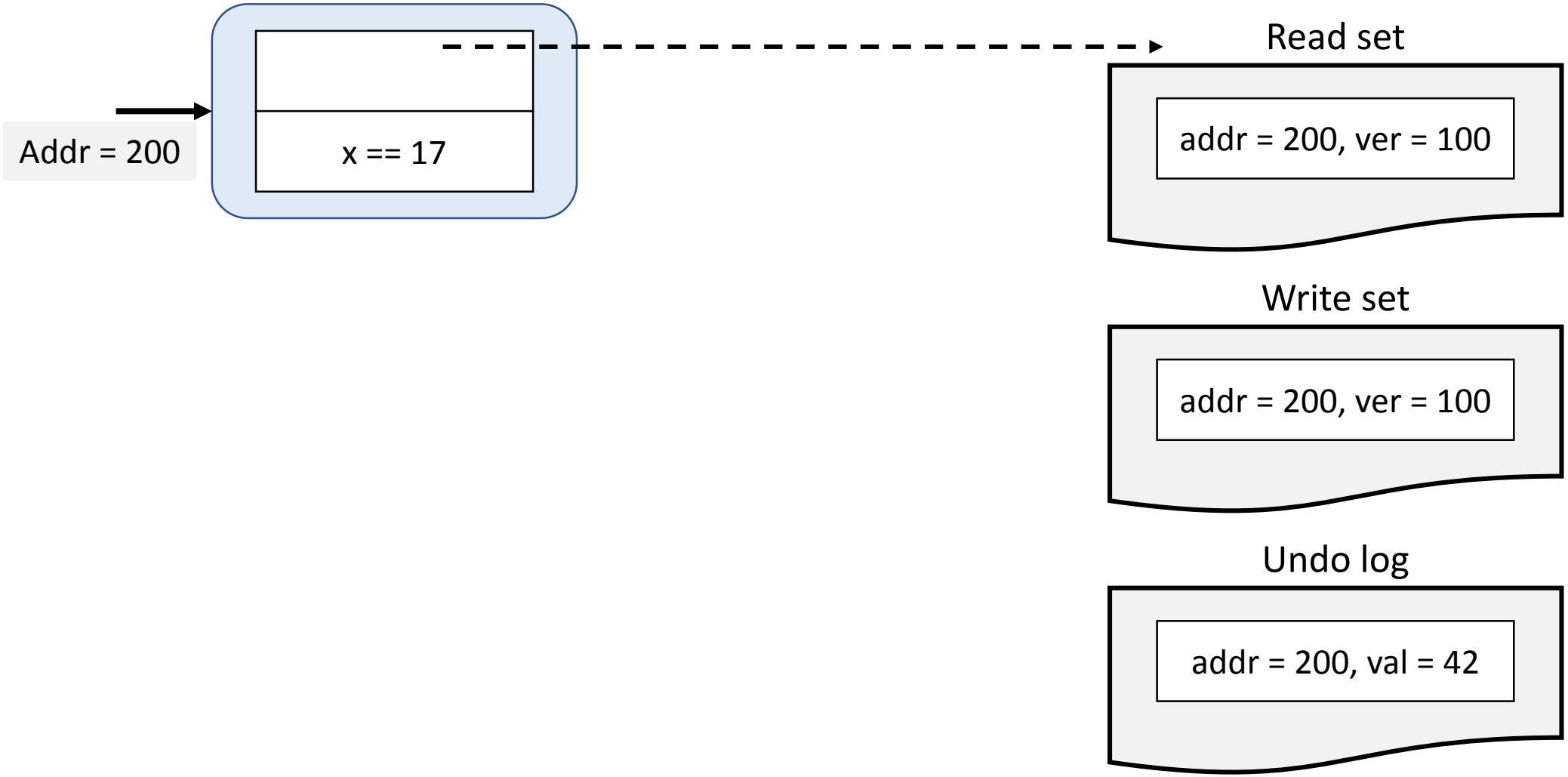
```
bool commitTx(tx) {
  foreach (entry e in tx.readSet)
    if (!validate(e.obj, e.ver))
      abortTx(tx);
      return false;
  foreach (entr e in tx.writeSet)
    unlock(e.obj, e.ver);
  return true;
}
```

Unlock increments the version number

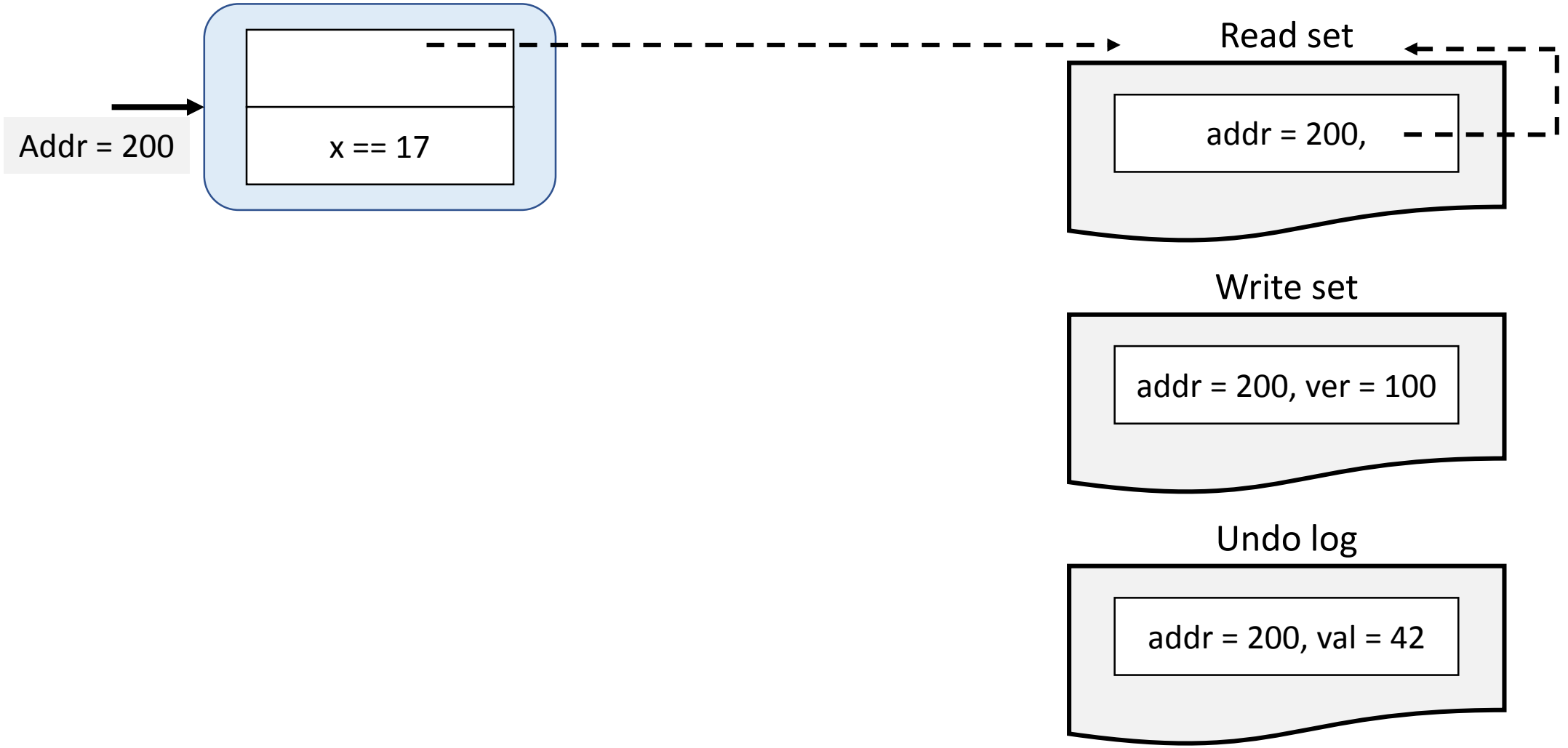
No Conflict on Read from Addr=200



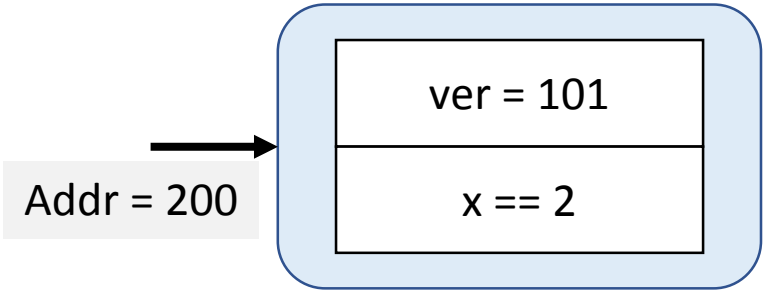
No Conflict on Read from and Write to Addr=200

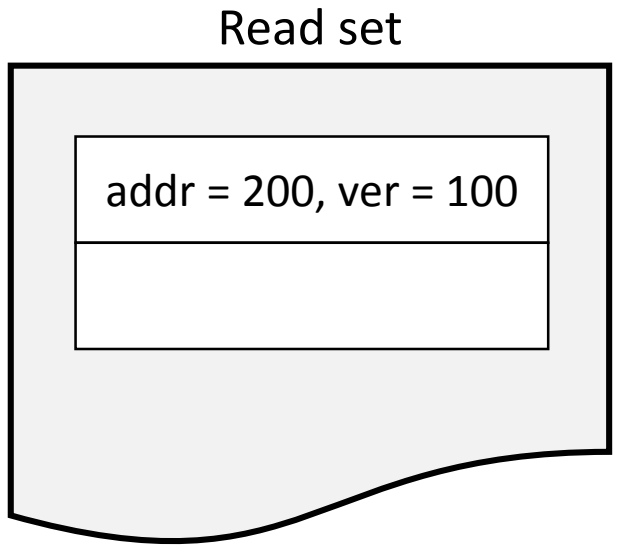


No Conflict on Write to and Read from Addr=200



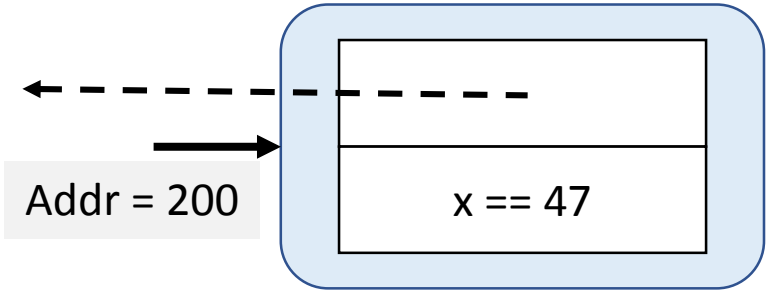
Conflict on Read from Addr=200, Concurrent Tx Updates and Commits

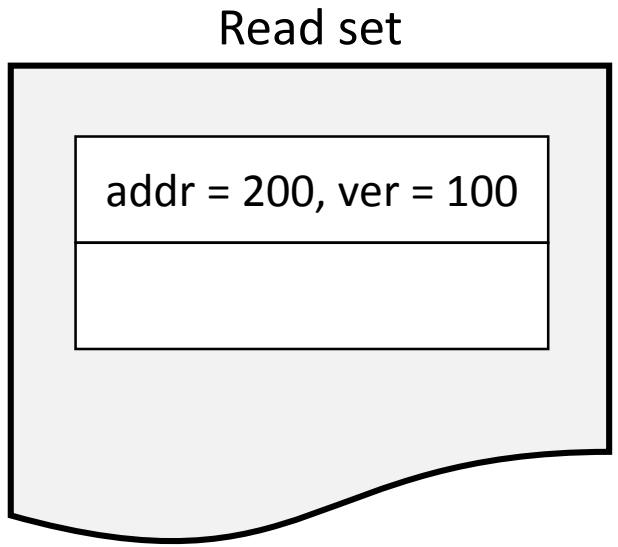




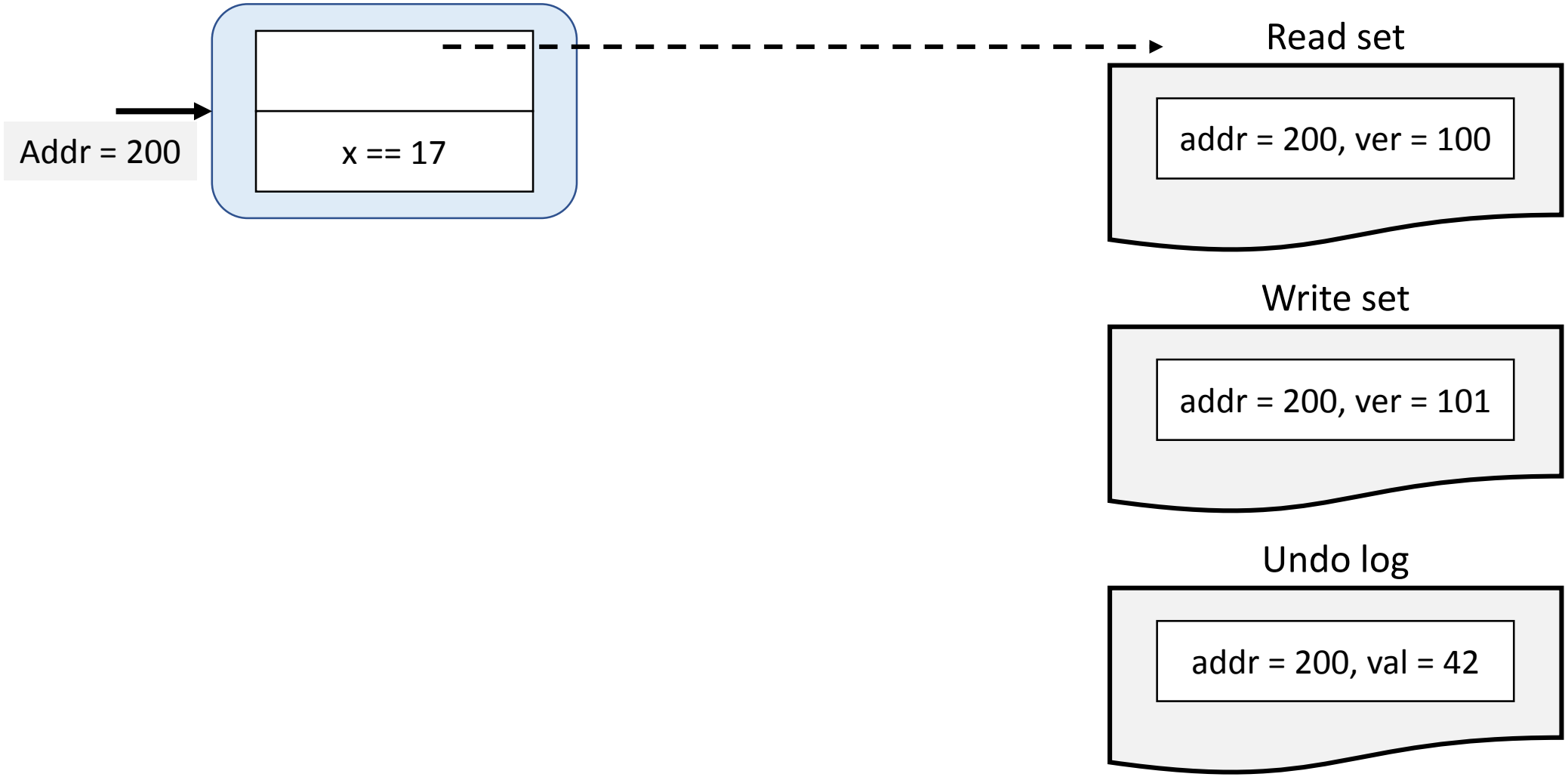
Conflict on Read from Addr=200, Concurrent Write

Conflict on Read from Addr=200 during Commit





Conflict Between Read and Write from Addr=200



Practical Issues

Version overflow

Do these techniques (McRT, Bartok) allow zombie txs?

Semantics of McRT and Bartok

- Read set may not remain consistent during txs
- Does not detect conflicts between txs and non-txs

Hardware Transactional Memory

Hardware Transactional Memory (HTM)

- Can provide strong isolation without modifications to non-Tx accesses

- TCC, ISCA'04
- LogTM, HPCA'06
- Rock HTM, ASPLOS'09
- FlexTM, ICS'09
- Azul HTM
- Intel TSX
- IBM Blue Gene/Q

Possible ISA Extensions

Explicit

- `begin_transaction`
- `end_transaction`
- `load_transactional`
- `store_transactional`

Implicit

- `begin_transaction`
- `end_transaction`

Which do you think is simpler?

Design Issues in HTMs

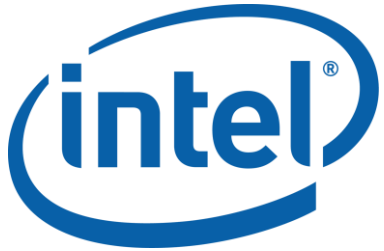
Tracking read and write sets

- Requires additional structures
- Extend existing data caches to track accesses
  - Granularity matters

Conflict detection

- Natural to piggyback on cache coherence protocols to detect conflicts

Intel Transactional Synchronization Extensions



TSX supported by Intel in selected series based on Haswell microarchitecture

TSX hardware can dynamically determine whether threads need to serialize lock-protected critical sections

---

https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell
https://software.intel.com/en-us/blogs/2012/02/07/coarse-grained-locks-and-transactional-synchronization-explained

Intel Transactional Synchronization Extensions

TSX operation

- Optimistically executes critical sections eliding lock operations
- Commit if the Tx executes successfully
- Otherwise abort – discard all updates, restore architectural state, and resume execution
- Resumed execution may fall back to locking

TSX Interface

Hardware Lock Elision (HLE)

- `xacquire`
- `xrelease`

- Extends HTM support to legacy hardware

Restricted Transactional Memory (RTM)

- `xbegin`
- `xend`
- `xabort`
- `xtest`

- New ISA extensions

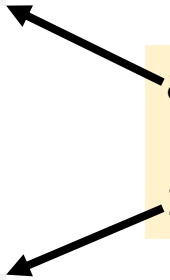
Hardware Lock Elision (HLE)

- Application uses legacy-compatible hints to identify critical sections
  - Hints ignored on hardware without TSX
- HLE provides support to execute critical section transactionally without acquiring locks
- Abort causes a re-execution without lock elision
- Hardware manages all state

Intel Transactional Synchronization Extensions. Intel Developer Forum 2012.

Lock Acquire Code

```
        mov eax, 1
Try:    lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```



```
acquire(mutex)
   /* critical section */
release(mutex)
```

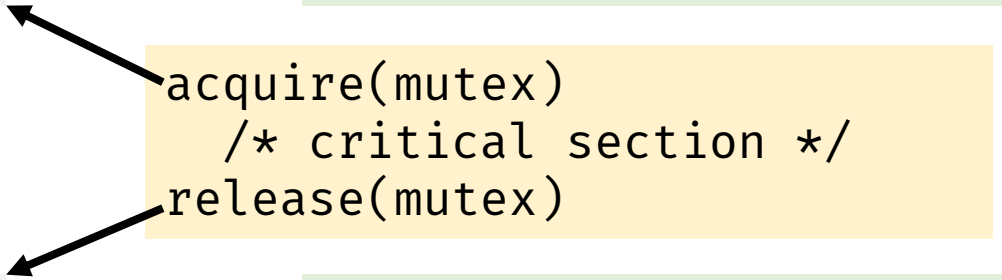
```
        mov mutex, 0
```

Intel Transactional Synchronization Extensions. Intel Developer Forum 2012.

HLE Interface

```
        mov eax, 1
Try:    lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

```
        mov eax, 1
Try:    xacquire lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```




```
        mov mutex, 0
```

```
        xrelease mov mutex, 0
```

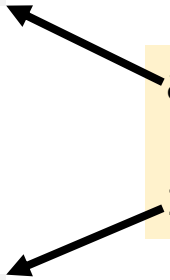
Intel Transactional Synchronization Extensions. Intel Developer Forum 2012.

Restricted Transactional Memory (RTM)

- Software uses new instructions to identify critical sections
- Similar to HLE, but more flexible interface for software
  - Requires programmers to provide an alternate fallback path
- Abort transfers control to target specified by XBEGIN operand
- Abort information encoded in the EAX GPR

Lock Acquire Code

```
        mov eax, 1
Try:    lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```



```
acquire(mutex)
   /* critical section */
release(mutex)
```

```
        mov mutex, 0
```

Intel Transactional Synchronization Extensions. Intel Developer Forum 2012.

RTM Interface

```
Retry:  xbegin Abort
        cmp mutex, 0
        jz Success
        xabort $0xff
Abort:
       // check eax and do retry policy
       // actually acquire lock or wait
to retry
       …
```

```
        mov eax, 1
Try:    lock xchg mutex, eax
        cmp eax, 0
        jz Success

Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

```
        cmp mutex, 0
        jnz Rel
        xend
```

```
Rel:    mov mutex, 0
```

Intel Transactional Synchronization Extensions. Intel Developer Forum 2012.

Aborts in TSX

- Conflicting accesses from different cores (data, locks, false sharing)
- Capacity misses
- Some instructions always cause aborts
  - System calls, I/O
- Eviction of a transactionally-written cache line
- Eviction of transactionally-read cache lines do not cause immediate aborts
  - Backed up in a secondary structure which might overflow

Section 12.2.4 in Intel 64 and IA-32 Architectures Optimization Reference Manual

Finding Reasons for Aborts can be Hard!

EAX register bit position	Meaning
0	Set if abort caused by XABORT instruction
1	If set, the transaction may succeed on a retry. This bit is always clear if bit 0 is set
2	Set if another logical processor conflicted with a memory address that was part of the transaction that aborted
3	Set if an internal buffer overflowed
4	Set if debug breakpoint was hit
5	Set if an abort occurred during execution of a nested transaction
23:6	Reserved
31:24	XABORT argument (only valid if bit 0 set, otherwise reserved)

TSX Implementation Details

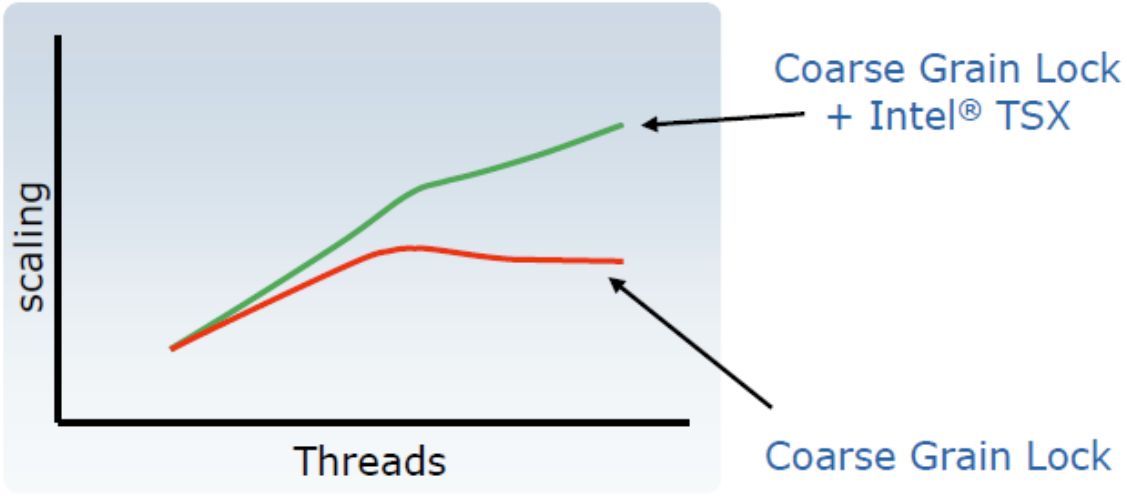
- Every detail is not known
  - Read and write sets are at cache line granularity
  - Uses L1 data cache as the storage
- Conflict detection is through cache coherence protocol

TSX Caveats

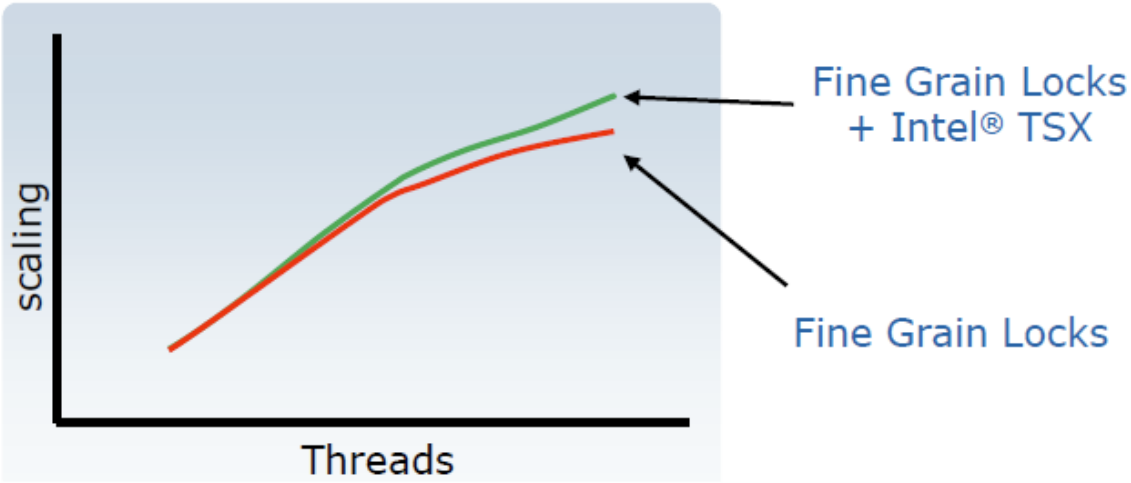
- No guarantees that txs will commit
- There should be a software fallback independent of TSX to guarantee forward progress

Applying Intel® TSX

Application with Coarse Grain Lock



Application re-written with Finer Grain Locks



An example of secondary benefits of Intel® TSX

Intel Transactional Synchronization Extensions. Intel Developer Forum 2012.

So what?

- GNU glibc 2.18 added support for lock elision of pthread mutexes of type PTHREAD_MUTEX_DEFAULT. Glibc 2.19 added support for elision of read/write mutexes
  - Depends whether the --enable-lock-elision=yes parameter was set at compilation time of the library
- Java JDK 8u20 onward support adaptive elision for synchronized sections when the -XX:+UseRTMLocking option is enabled
- Intel Thread Building Blocks (TBB) 4.2 supports elision with the speculative_spin_rw_mutex

References

- T. Harris et al. – Transactional Memory, 2nd edition.
- D. Sorin et al. – A Primer on Memory Consistency and Cache Coherence
- R. Yoo et al. - Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. SC 2013.
- Intel 64 and IA-32 Architectures Optimization Reference Manual