CS 636: Transactional Memory

Swarnendu Biswas

Semester 2020-2021-II

CSE, IIT Kanpur

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

Challenges with Concurrent Programming

Order, atomicity & sequential consistency violations

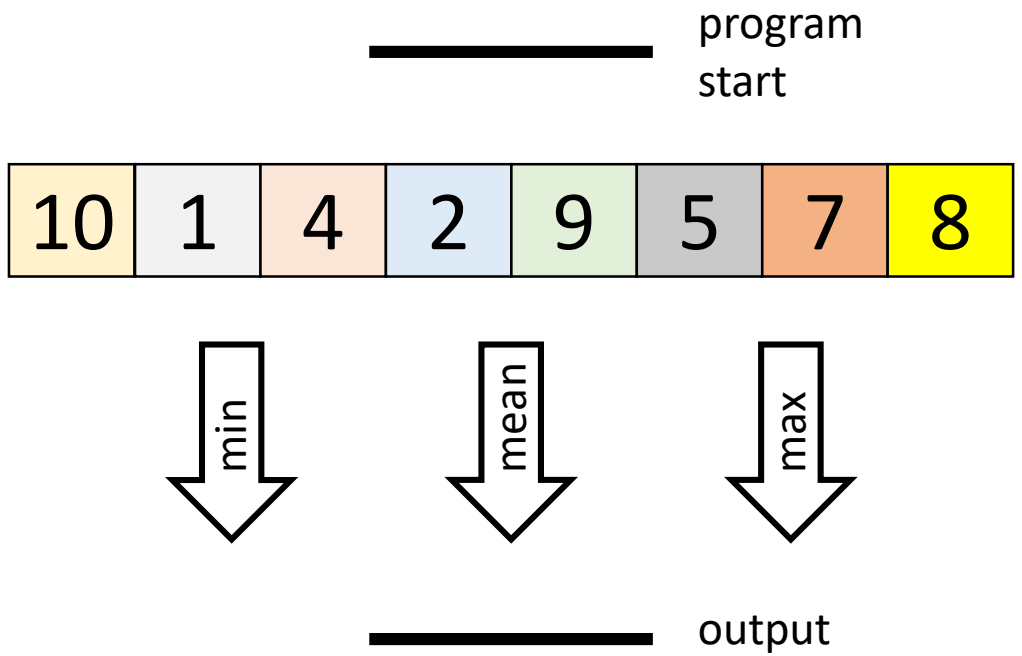
Concurrent and correct

Deadlock

Poor performance: lock contention, serialization

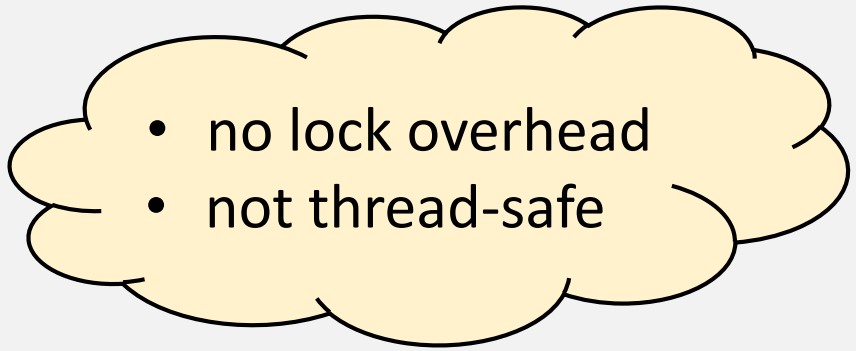
Task Parallelism

- Different tasks run 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 to find bucket
  HashEntry e = buckets[idx];
  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);
  }
}
```

- Thread-safe, uses explicit coarse-grained locking

Coarse-Grained and Fine-Grained Locking

Coarse-grained

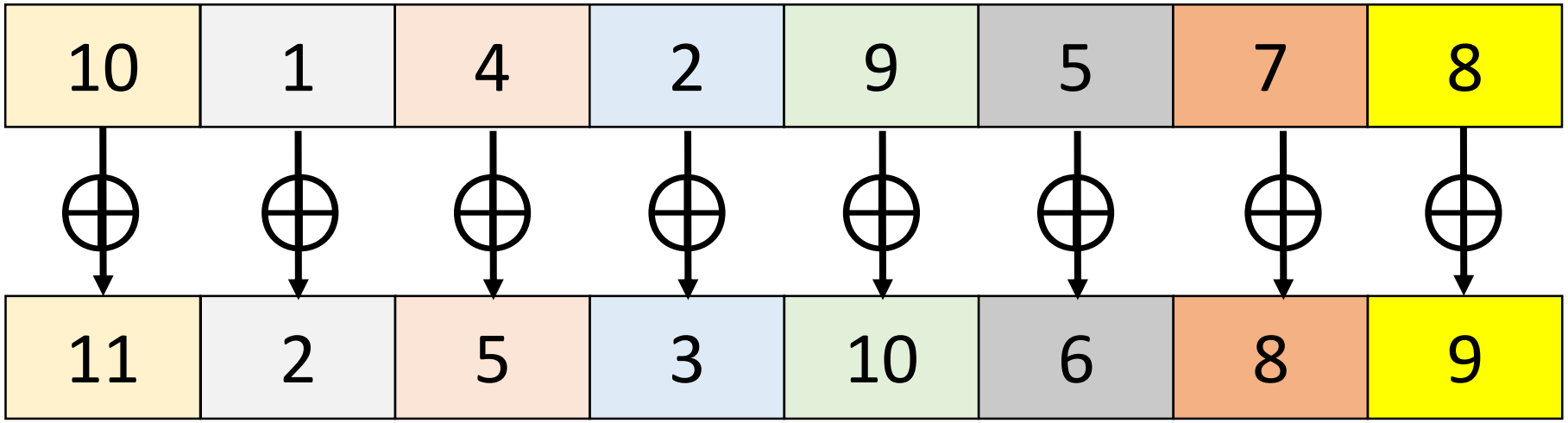
- **Pros**: Easy to implement
- **Cons**: limits concurrency, poor scalability

Fine-grained

- **Idea**: Use a separate lock per bucket
- **Pros**: thread safe, more concurrency, better performance
- **Cons**: difficult to get correct, more 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 • Speedup is usually less since it may require synchronization	• Same operation on different data • Parallelization proportional to the input data size • Speedup is usually more

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

Programming languages provide abstraction and composition

- Procedures, ADTs, and libraries

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

Abstraction and Composability

- Parallel programming lacks abstraction mechanisms
  - Low-level parallel programming models, such as threads and explicit synchronization, are unsuitable for constructing abstractions
  - Explicit synchronization is not composable

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
- Lock convoying
- 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

```
/*
 * 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

Lock-based Synchronization is not Composable

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

You want to add a new method:

```
boolean move(HashTable tab1, HashTable tab2, T elem)
      => remove()
      => insert()
```

Lock-based Synchronization is not Composable

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

You v

```
-> insert()
```

- Option: Add new methods such as `LockHashTable()` and `UnlockHashTable()`
  - Breaks the abstraction by exposing an implementation detail
- Lock methods are error prone
  - A client that locks more than one table must be careful to lock them in a globally consistent order to prevent deadlock

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

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

Easy programmability (like coarse-grained locks)

- Performance goal is like fine-grained locks

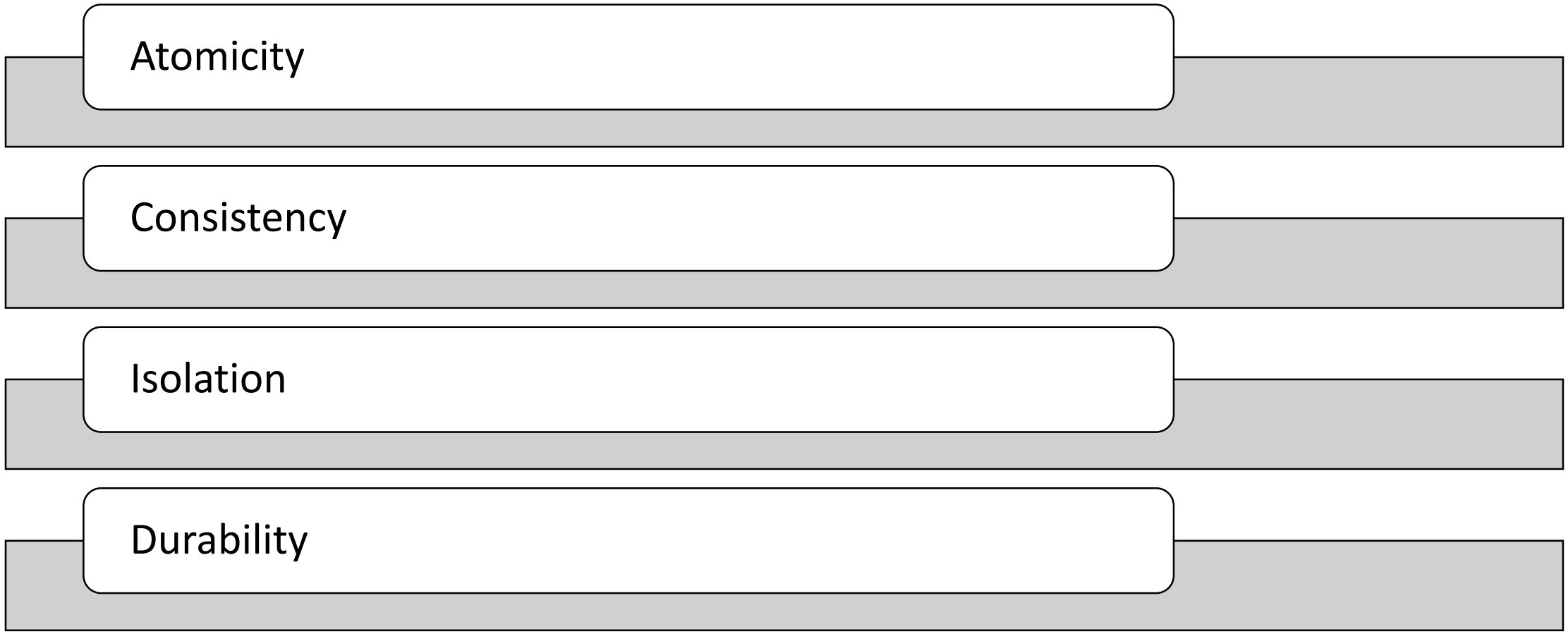
Basic TM Design

- 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 writes 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

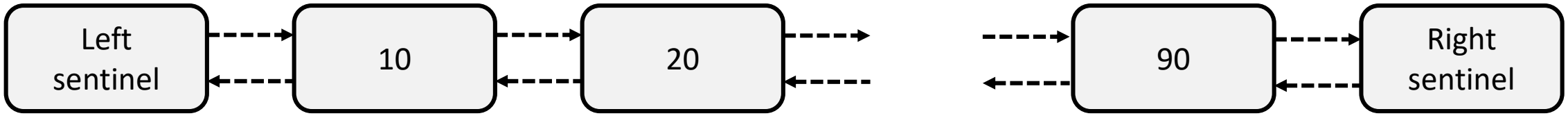
Thread 2

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

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

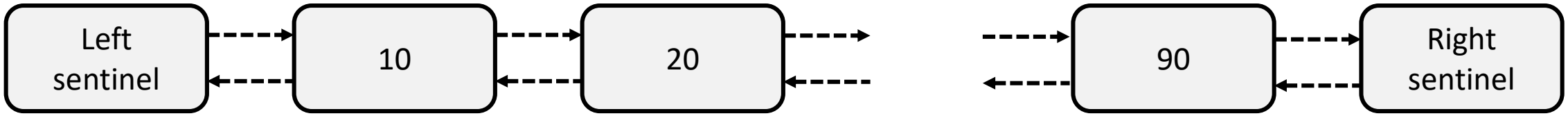
No data race due to TM semantics

Linked-List-based Double Ended Queue




```
void PushLeft(DQueue *q, int val) {
  QNode *qn = malloc(sizeof(QNode));
  qn->val = val;
  atomic {
    QNode *leftSentinel = q->left;
    QNode *oldLeftNode = leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
  }
}
```

Linked-List-based Double Ended Queue



- Challenges with a lock-based implementation
  - A single lock would prevent concurrent operations at both ends
  - Need to be careful to avoid deadlocks with multiple locks
  - Take care of corner cases (for example, only one element is left)

```
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
  }
}
```

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

- Thread-safe, easy to program
- No lock-related issues

Cons

- Good performance and scalability depends on the TM implementation

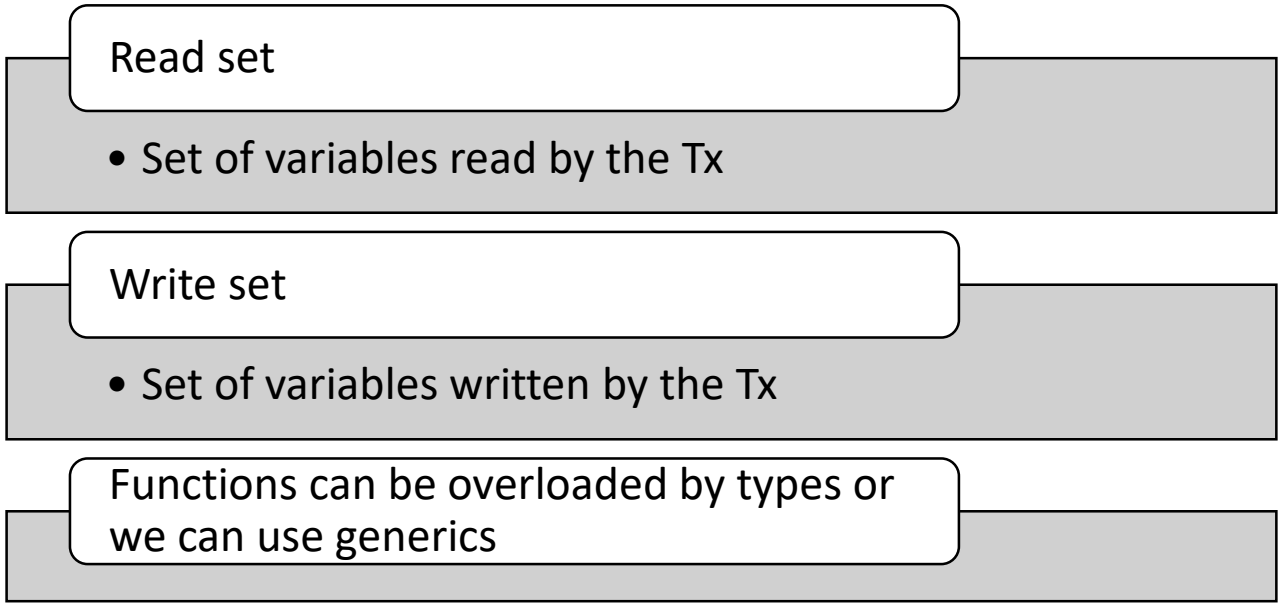
`synchronized` in Java

`synchronized`	TM Transaction
• Provides mutual exclusion compared to other blocks on the same lock	• A transaction is atomic w.r.t. all other transactions in the system
• Nested blocks can deadlock if locks are acquired in wrong order	• Nested transactions never deadlock

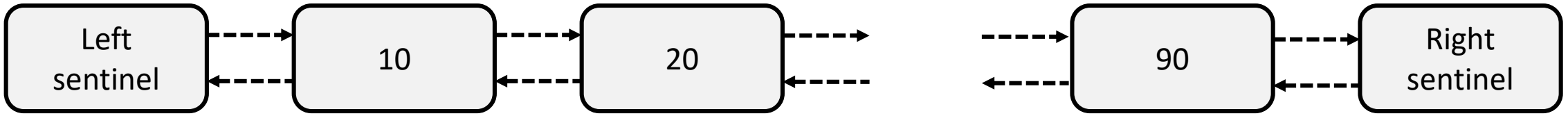
TM Interface

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

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



Linked-List-based Double Ended Queue




```
void PushLeft(DQueue *q, int val) {
  QNode *qn = malloc(sizeof(QNode));
  qn->val = val;
  do {
    StartTx();
    QNode *leftSentinel = ReadTx(&(q->left));
    QNode *oldLeftNode = ReadTx(&(leftSentinel->right));
    WriteTx(&(qn->left), leftSentinel);
    WriteTx(&(qn->right), oldLeftNode);
    WriteTx(&(leftSentinel->right), qn);
    WriteTx(&(oldLeftNode->left), qn);
  } while (!CommitTx());
}
```

- Similar to sequential code
- No explicit locks

Transactions cannot replace all uses of locks!

Thread 1

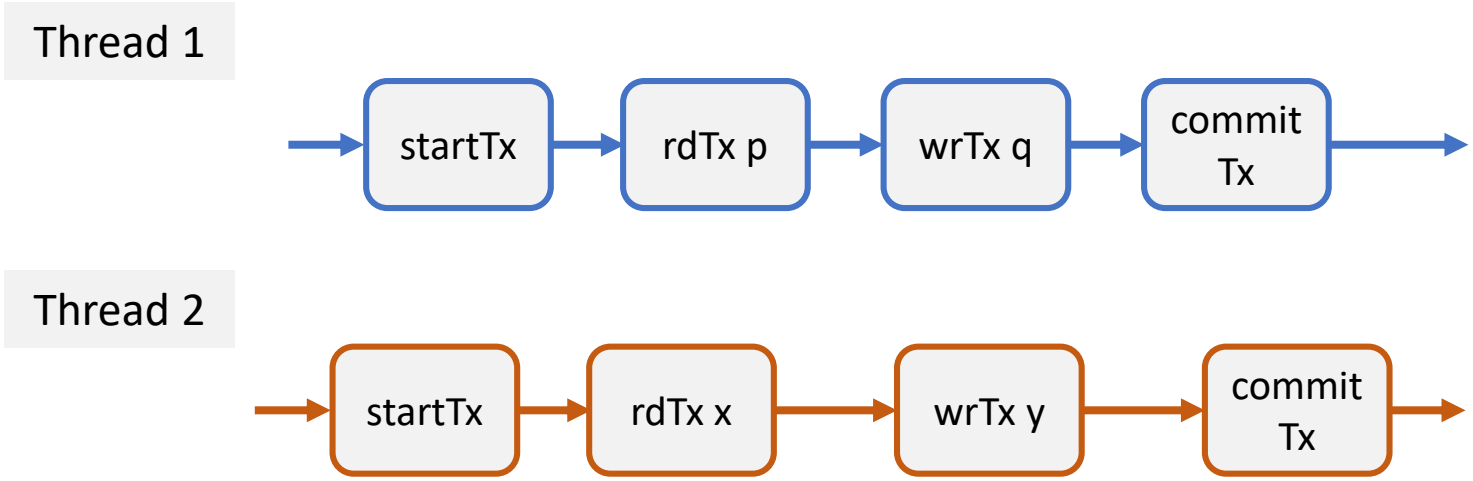
```
do {
  startTx();
  writeTx(&x, 1);
} while (!commitTx());
```

Thread 2

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

Concurrency in TM

- Two levels
  - Among Txs from concurrent thread
  - Among individual Tx operations



Design Choices

- Concurrency Control
- Version Management
- Conflict Detection

TM Terminology

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

Let R_i and W_j be the read and write sets of Tx i. Then a conflict occurs if and only if

- $R_i \cap W_j \neq \emptyset$, or
- $W_i \cap W_j \neq \emptyset$, or
- $W_i \cap R_j \neq \emptyset$

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

- Delay or abort one of the conflicting transactions

A conflict, its detection, and its resolution can occur at different times

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

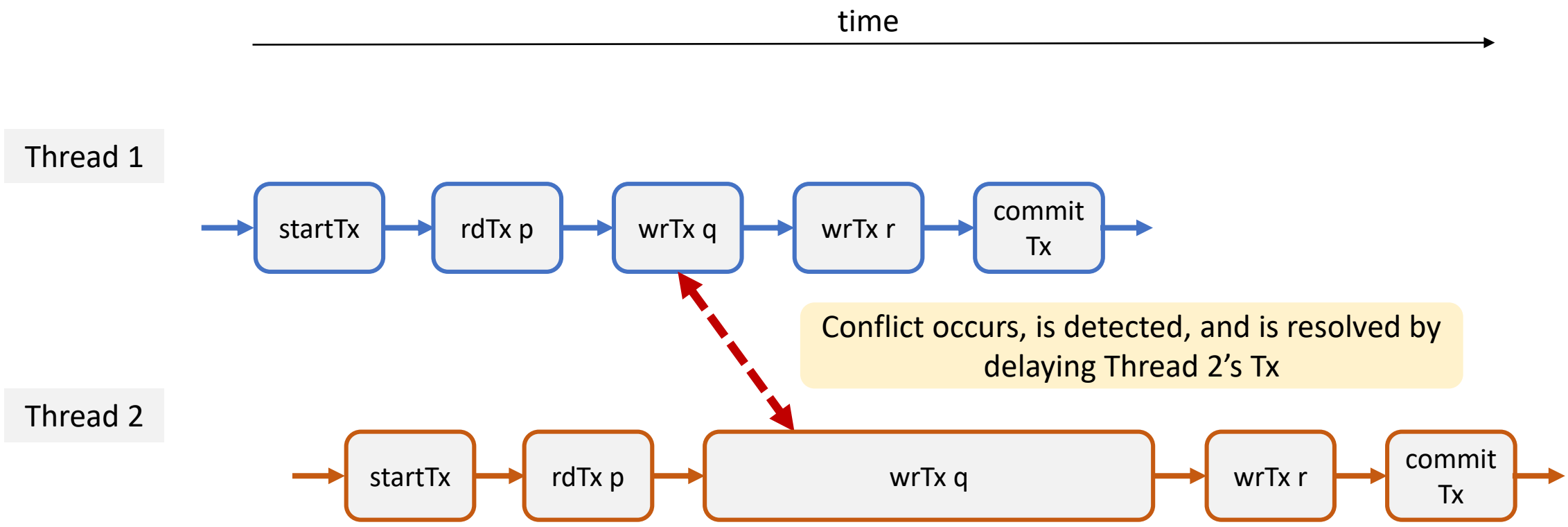
Pessimistic

- Occurrence, detection, and resolution happen **at the same time** during execution
- Claims ownership of data before modifications

Optimistic

- Conflict detection and resolution can happen **after** the conflict occurs
- Multiple conflicting transactions can continue to keep running, as long as the conflicts are detected and resolved before the Txs commit

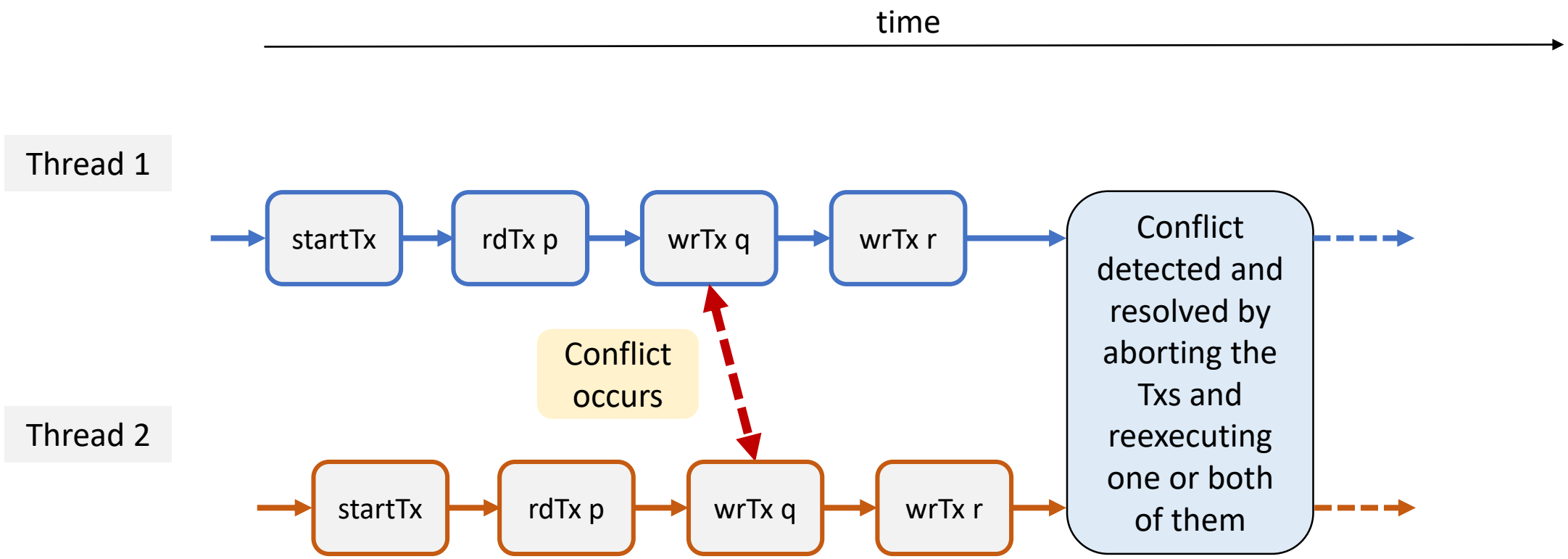
Pessimistic Concurrency Control



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	• Avoids claiming exclusive ownership of data, provides more conflict resolution choices
• Effective in high contention cases	• Effective in low contention cases
• Needs to avoid or detect and recover from deadlock situations	• Needs to avoid livelock situations through contention management schemes

Hybrid Concurrency Control

Use pessimistic control for writes and optimistic control for reads

Use optimistic control TM with pessimistic control of irrevocable Txs

- Irrevocable Tx means that the changes cannot be rolled back
- A Tx that has performed I/O or a Tx that has experienced frequent conflicts in the past

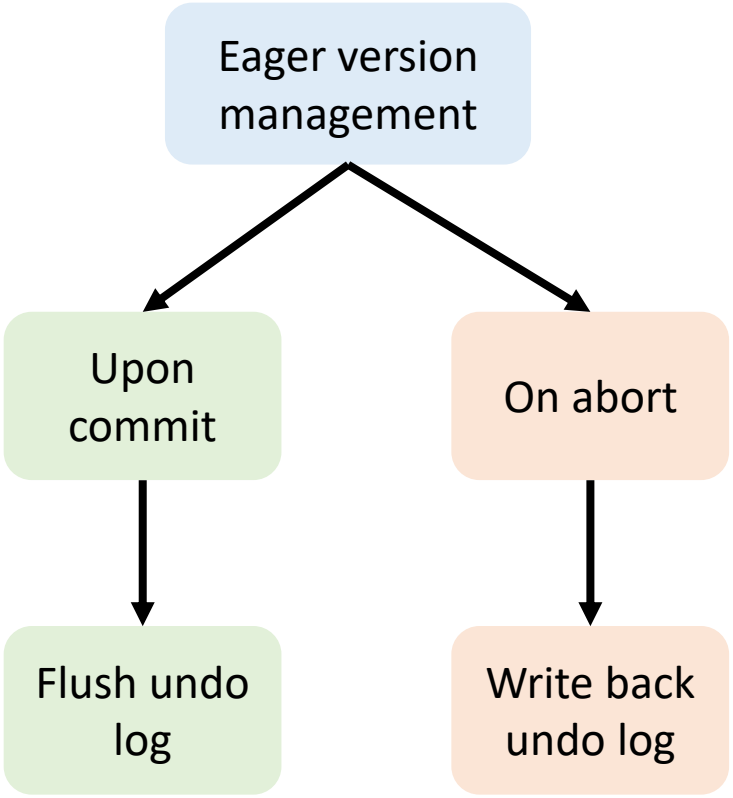
Version Management

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

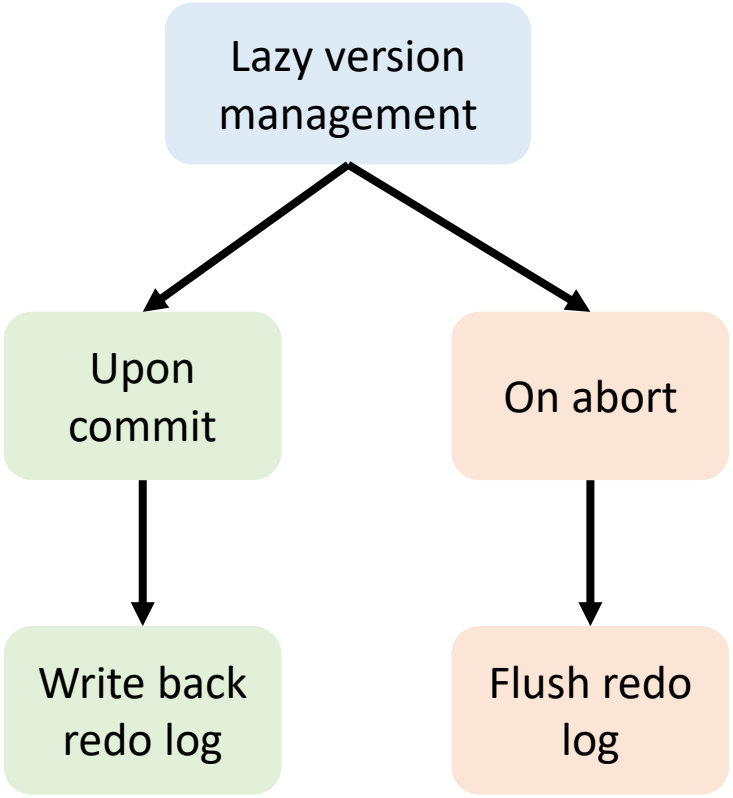
Which concurrency control type should we use, pessimistic or optimistic?



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 in software TM, line offset or whole cache line in hardware TM
- What are the tradeoffs?

Time of conflict detection

- Just before access (eager), during validation, during final validation before commit (lazy)
- Validation can occur at any time, and can occur multiple times

Conflicting access types

- Among concurrent ongoing 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
                                            .
   ARRAY LAYOUT:                            .
|<---------- array header ----------------->|
+----------+------------+------------+------+------+------+------+------+
| GCHeader | MiscHeader | JavaHeader | len  | elt0 | elt1 | ...  |eltN-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

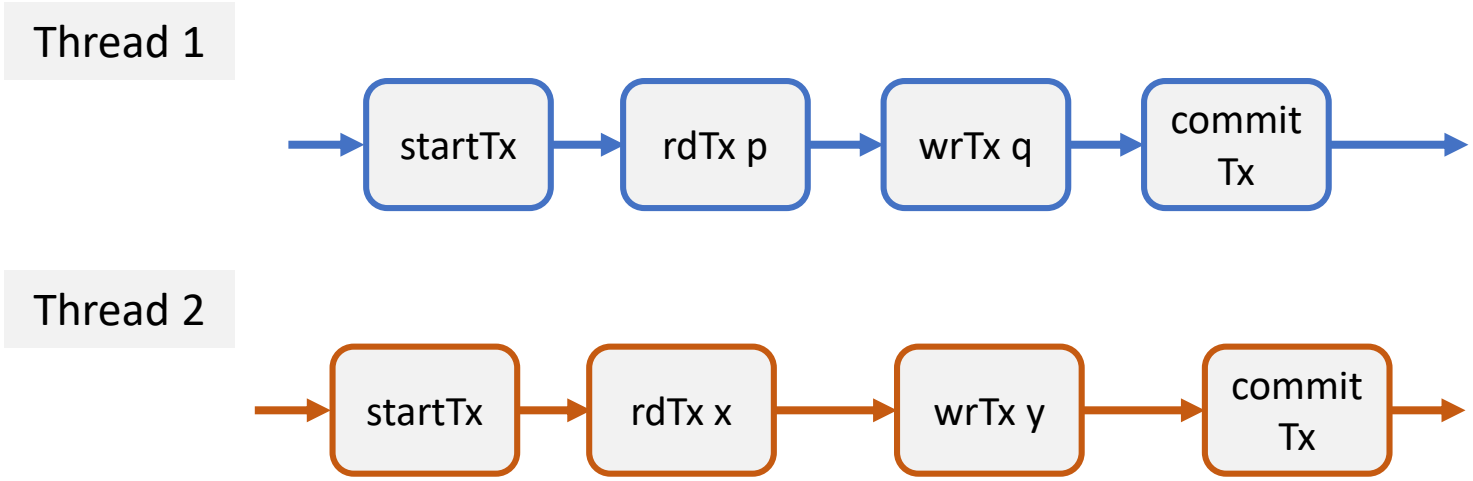
```
…
y = 20;
…
```

- Detect conflicts at the granularity of objects or fields
- A hardware technique can detect conflicts at the line/block level or at the level of individual byte offsets
- What are the tradeoffs?

Transaction Semantics

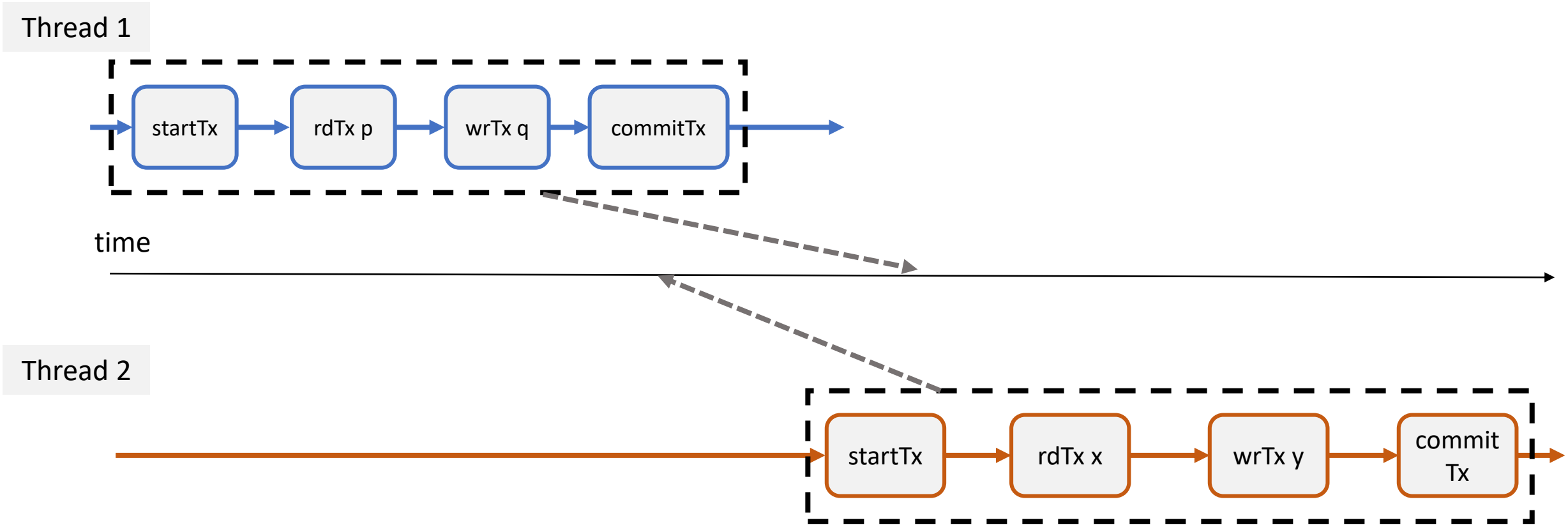
Concurrency in TM

- Two levels
  - Among Txs from concurrent thread
  - Among individual Tx operations



Serializability

The result of executing concurrent transactions must be identical to *a* result in which these transactions executed serially

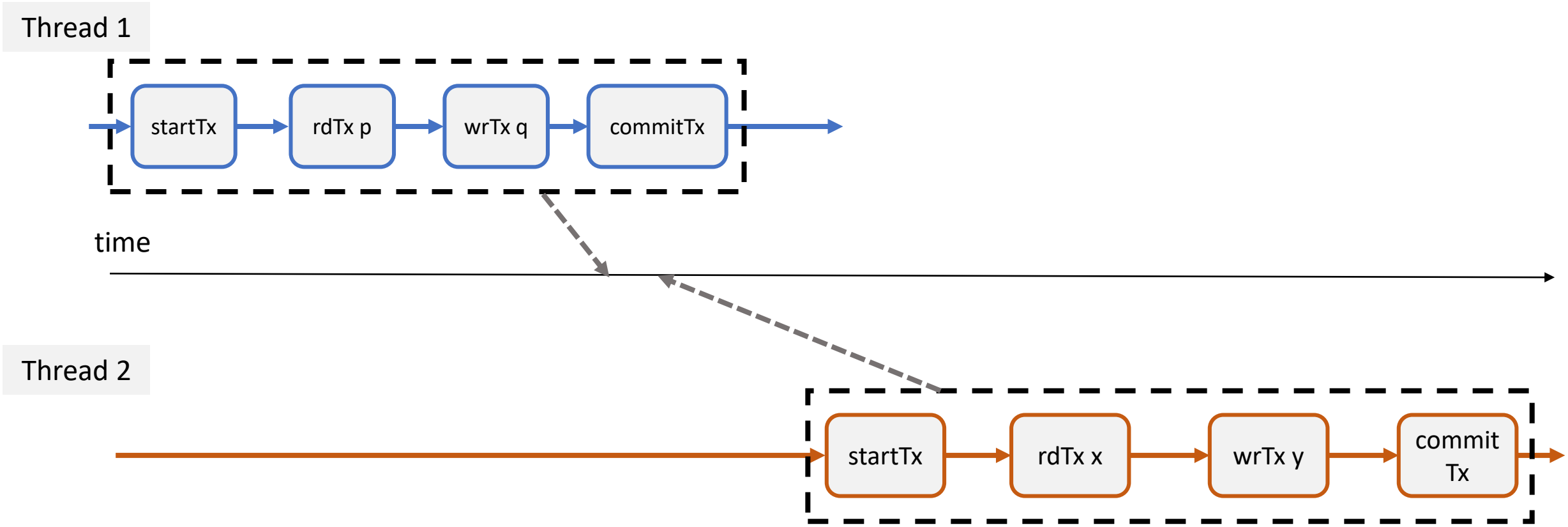


Serializability

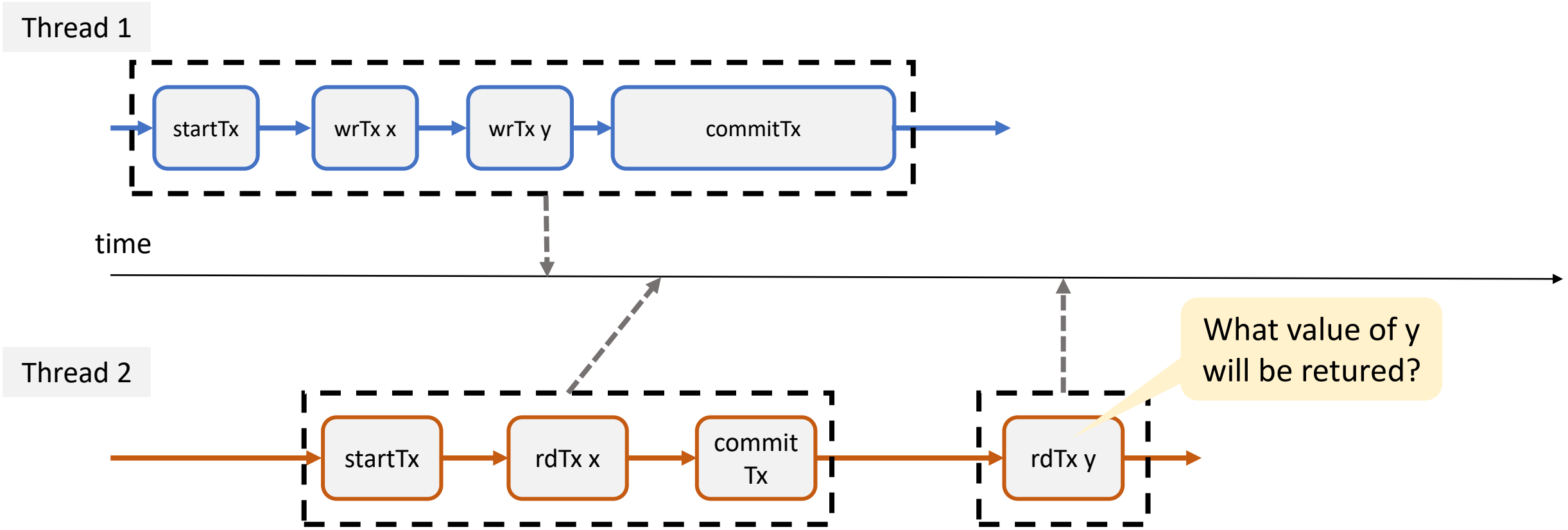
- Widely-used correctness condition in databases
- The TM system can reorder transactions
- Serializability requires the Txs **appear** to run in serial order
  - Does not require that the order has to be real-time

- Strict serializability
  - If transaction TA completes before transaction TB starts, then TA must occur before TB in the equivalent serial execution

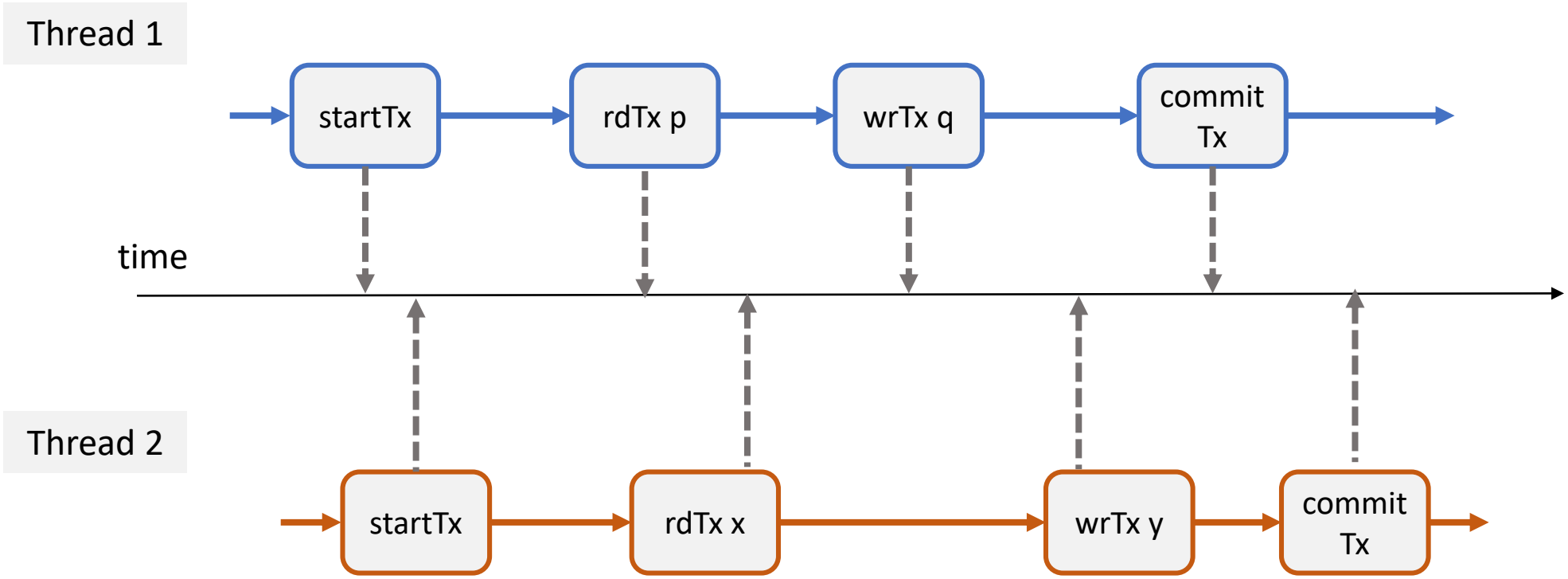
Strict Serializability



Limitations of Strict Serializability



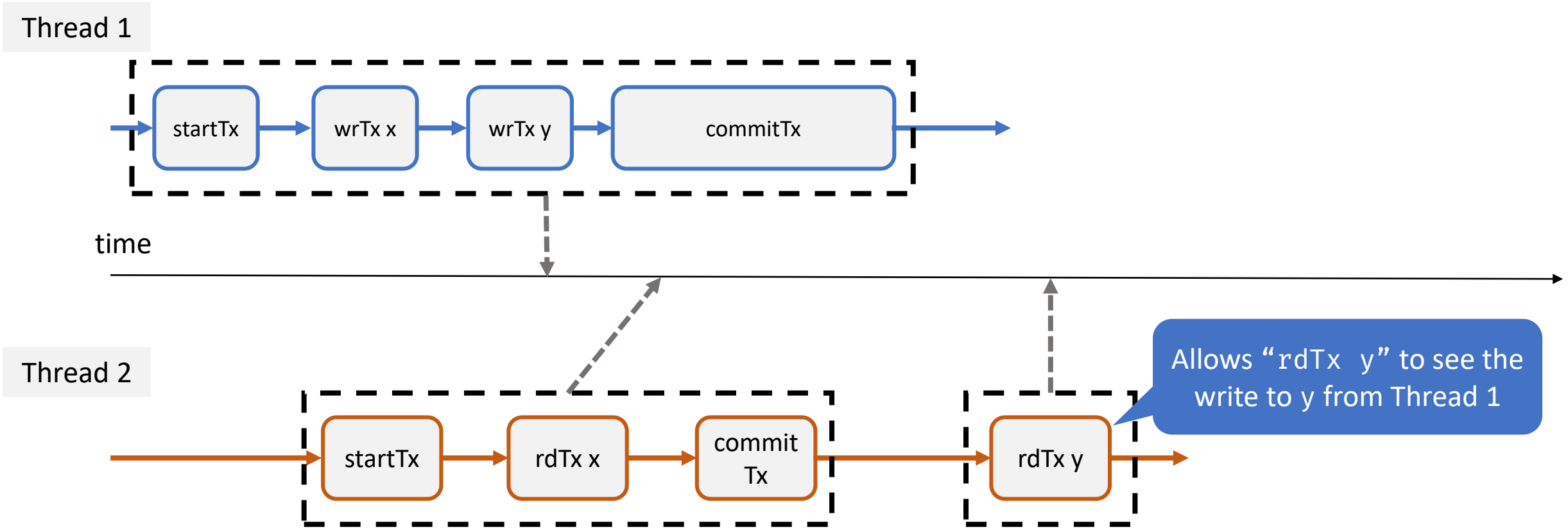
Linearizability



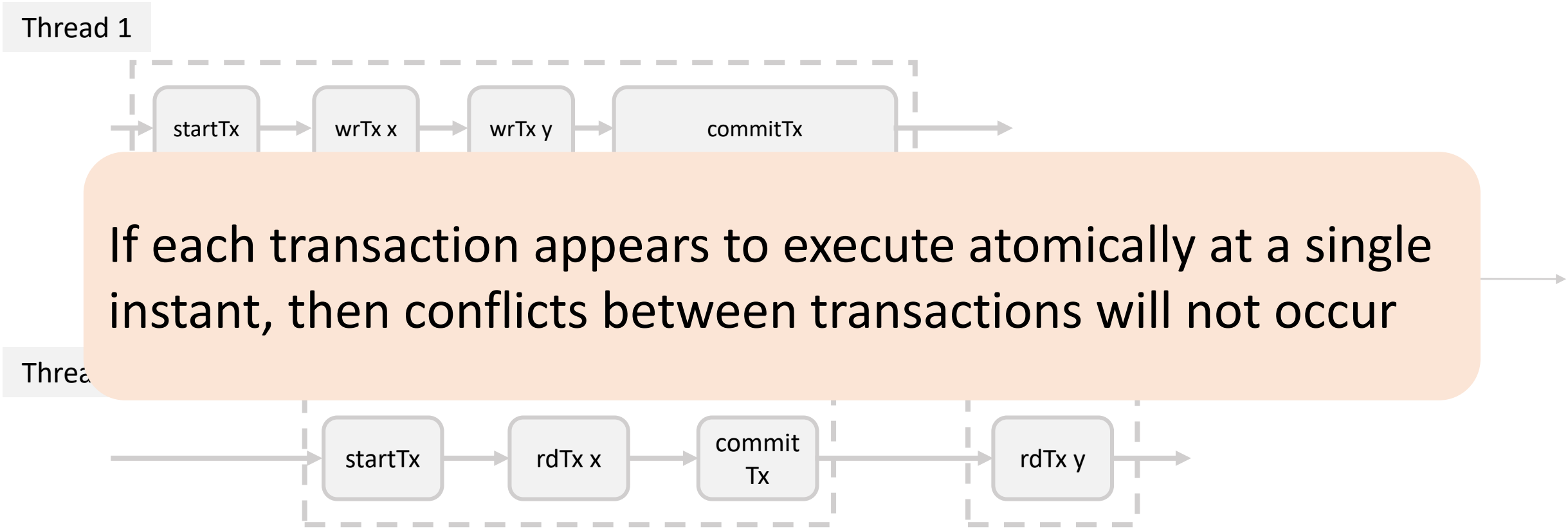
Linearizability

- A method call is the interval that starts with an invocation event and ends with a response event
  - A method call is pending if the response event has not yet occurred
- Linearizability of an operation: each operation appears to execute atomically at some point between its invocation and its completion
- Linearizability of a transaction: a transaction is a single operation extending from the beginning of `startTx()` until the completion of its final `commitTx()`

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());
```

What are possible values of x and y after execution?

- With serializability
- With SI

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 Exceptions Under Strong Memory Models. ISMM 2017.

Understanding SI

- Semantics of SI may seem unexpected when compared with simpler models based on serial ordering of complete transactions
- Potential increased concurrency often does not manifest as a performance advantage when compared with models such as strict serializability

Other TM Considerations

Consistency During Transactions

- Semantics such as serializability characterize the behavior of committed Txs
- What about the Txs which fail to commit?
  - Tx may abort or may be slow to reach `commitTx()`

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());
```

Assume eager version management and lazy conflict detection

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());
```

Assume eager version management and lazy conflict detection

Validation only during commit is insufficient for this TM design

Considerations with Zombie Txs

- A Tx that is inconsistent but is not yet detected is called a **zombie** Tx
- Careful handling of zombie Txs are required, especially for unsafe languages like C/C++
  - Inconsistent values can potentially be used in pointer arithmetic to access unwanted memory locations
- Possible workarounds: perform periodic validations
  - Increases run-time overhead, validating n locations once requires n memory accesses
  - Couples the program to the TM system
    - A TM using eager updates allows a zombie transaction's effects to become visible to other transactions
    - A TM using lazy updates only allows the effects of committed transactions to become visible

Challenges with Mixed-Mode Accesses

- TM semantics must consider the interaction between transactional and non-transactional memory accesses
- Many TMs do not detect conflicts between transactional and non-transactional accesses
  - Can lead to unexpected behavior with zombie Txs
- Requires the non-Tx thread to participate in conflict detection

Challenges with Mixed-Mode Accesses

Weak atomicity

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

Strong atomicity

- Guarantees Tx semantics among Txs and non-Txs

Often referred to as weak and strong isolation (inspired by databases)

Think of Challenges with Weak Atomicity

- Data races between Tx and non-Tx code
- Mismatched conflict detection granularity
  - Tx detects conflicts at a coarser granularity
- Complicated sharing idioms
  - Use a Tx to initialize shared data, expect other threads to read the data transactionally

Lock-Based Synchronization

```
java.util.LinkedList list is shared
```

```
Initially list == [Item{val1==0,val2==0}]
```

Thread 1

```
Item item;
synchronized(list) {
  item = list.removeFirst();
}
int r1 = item.val1;
int r2 = item.val2;
```

Thread 2

```
synchronized(list) {
  if (!list.isEmpty()) {
    Item item = list.getFirst();
    item.val1++;
    item.val2++;
  }
}
```

T. Shpeisman et al. Enforcing Isolation and Ordering in STM. PLDI 2007.

Can we safely replace `synchronize` with `atomic`?

`java.util.LinkedList list` is shared

`Initially list == [Item{val1==0,val2==0}]`

Thread 1

```
Item item;
weakly_atomic(list) {
  item = list.removeFirst();
}
int r1 = item.val1;
int r2 = item.val2;
```

Thread 2

```
weakly_atomic(list) {
  if (!list.isEmpty()) {
    Item item = list.getFirst();
    item.val1++;
    item.val2++;
  }
}
```

Consider a TM design with eager versioning and lazy conflict detection!

T. Shpeisman et al. Enforcing Isolation and Ordering in STM. PLDI 2007.

Few Issues to Consider with Weak Isolation

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

Thread 1	Thread 2

atomic {
  r1 = x;
 	
  r2 = x;
}
	x = 1;

- A non-repeatable read can occur if a Tx reads the same variable multiple times, and a non-Tx write is made to it in between
- Unless the TM buffers the value seen by the first read, the transaction will see the update

Initially x = 0

Thread 1	Thread 2

atomic {
  r = x;
 	
  x = r+1;
}
	x = 10;

- An intermediate lost update can occur if a non-Tx write interposes in a transactional read-modify-write sequence; the non-Tx write can be lost, without being seen by the Tx read

Initially x is even

Thread 1	Thread 2

atomic {
  x++;
 	
  x++;
}
	r = x;

- An intermediate dirty read can occur with a TM using eager version management in which a non-Tx read sees an intermediate value written by a transaction, rather than the final, committed value

Single-Lock Atomicity for Transactions

- How do we provide semantics for mixed-mode accesses?
- A program executes as if all transactions acquire a single, program-wide mutual exclusion lock

Thread 1	Thread 2

`startTx();`
`while (True) {}`
`commitTx();`
	`startTx();`
	`int tmp = readTx(&x);`
	`commitTx();`

What will happen here with SLA?

- There are many other proposed models like DLA and TSC

Nested Transactions

- Nested parallelism is important
  - Utilizes increasing number of cores
  - Integrates with programming models like OpenMP
- Execution of a nested Tx is wholely contained in the dynamic extent of another Tx
- Many choices on how nested Txs interact
  - Flattened
    - Aborting the inner Tx causes the outer Tx to abort
    - Committing the inner Tx has no effect until the outer Tx commits
  - Closed
    - Inner Tx can abort without terminating its parent Tx

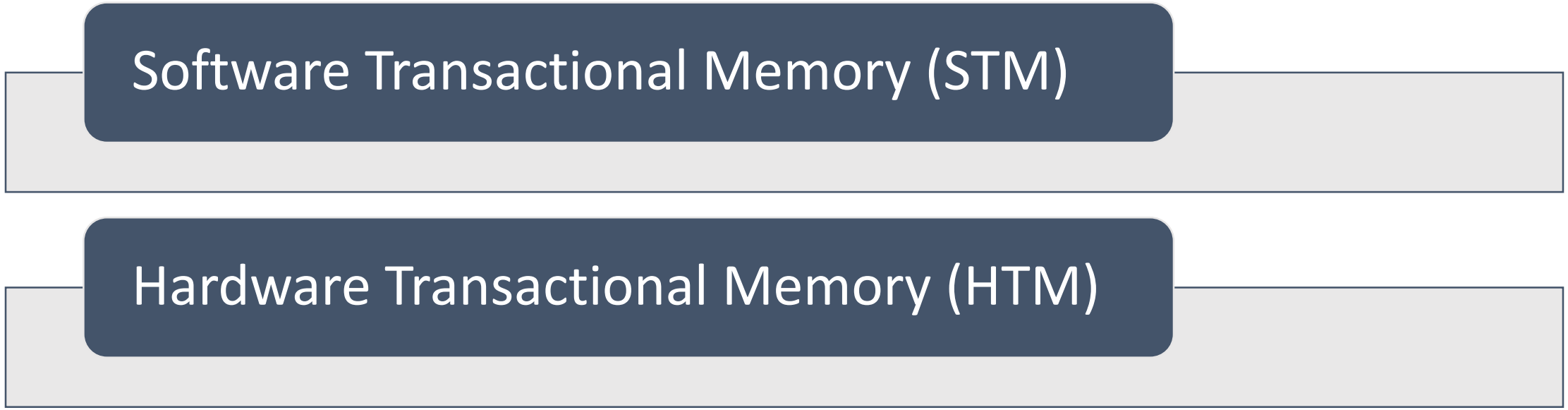
```
// Parallelize loops
FOR I := …
    FOR J := …
        FOR K := …
```

```
int x = 1;

do {
  StartTx();
  WriteTx(&x, 2);

  do {
    StartTx();
    WriteTx(&x, 3);
    AbortTx();
...
```

Providing Txs: TM Implementations



STMs vs HTMs

STM	HTM
• Supports flexible techniques in TM design	• Restricted variety of implementations
• Easy to integrate STMs with PL runtimes	• Need to adapt existing runtimes to make use of HTM
• Easier to support unbounded Txs with dynamically-sized logs	• Limited by bounded-sized structures like caches
• More expensive than HTMs	• Better performance than STMs

Software Transactional Memory

Swarnendu Biswas

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!

Remember well-designed applications should have low conflict rates

Is the design of undo log important in a TM with eager version management?

Is the design of redo log important in a TM with lazy version management?

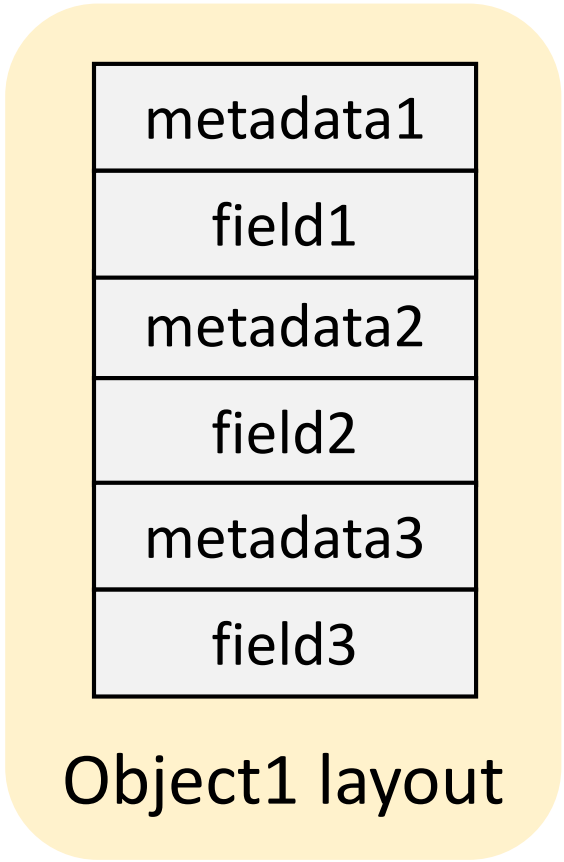
Implementing STM

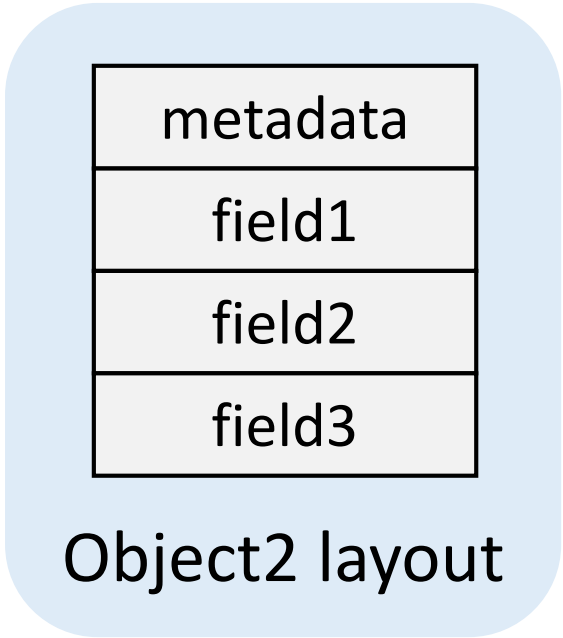
- Use compilation passes to instrument the program
  - `startTx()` – Tx entry point (prolog)
  - `commitTx()` – Tx exit point (epilog)
  - `readTx()/writeTx()` – Transactional read/write accesses
- TM runtime tracks memory accesses, detects conflicts, and commits/aborts Txs

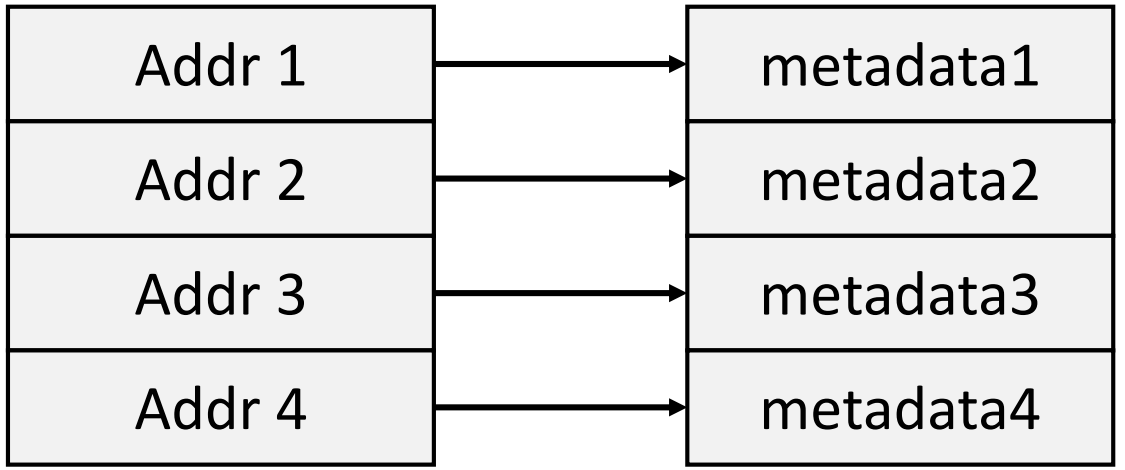
```
atomic {
  tmp = x;
  y = tmp + 1;
}
```

```
// Per-TX data structure
td = getTxDesc(thr);
startTx(td);
tmp = readTx(&x);
writeTx(&y, tmp+1);
commitTx(td);
```

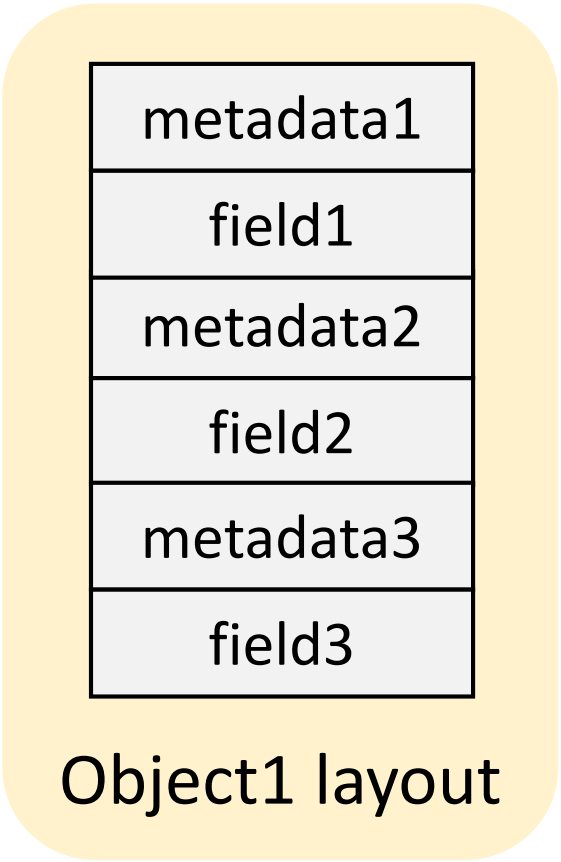
Object Metadata and Word Metadata

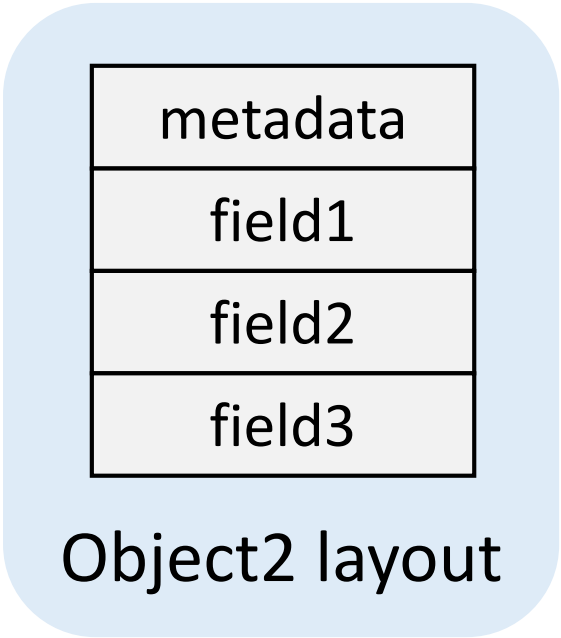






Pros and Cons of Metadata in Object Header





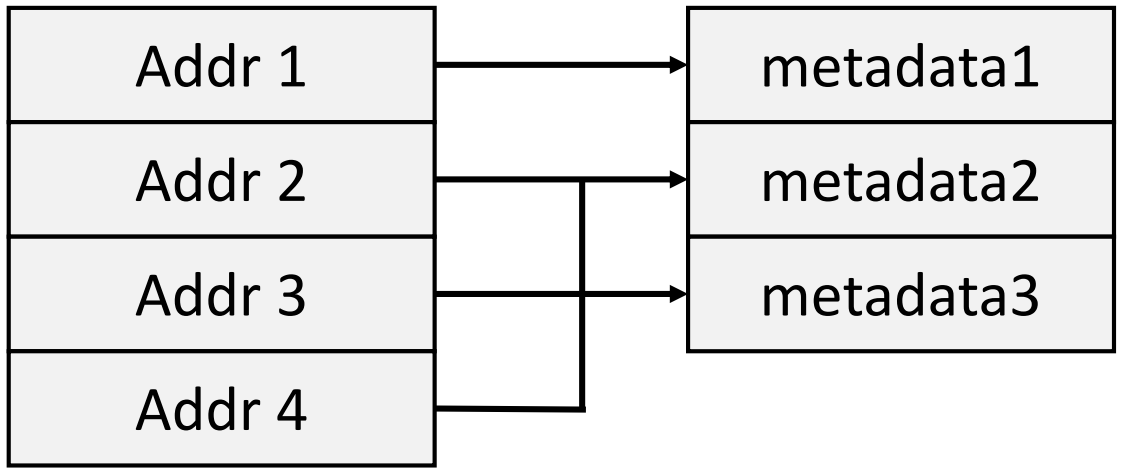
Pros

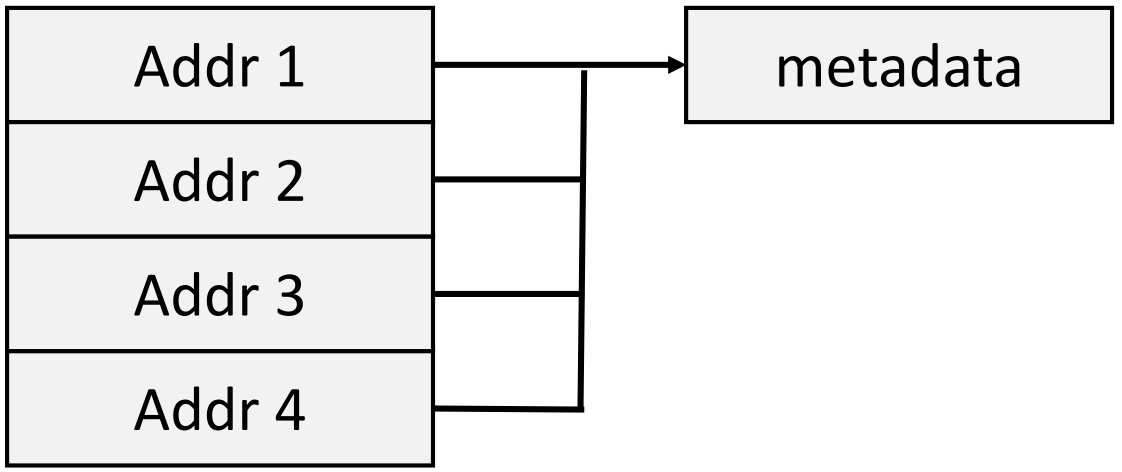
- May lie on the same cache line
- Single update for accesses to all fields

Cons

- Potential for false conflicts
- Increases coupling
  - GC considerations

Variants of Word-based Metadata





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

Process-wide metadata space

Which granularity to use?

Potential impact due to false conflicts

Impact on memory usage

Impact on performance

- Speed of mapping location to metadata

Major STM Designs

Per-object versioned locks (McRT-STM, Bartok-STM)

- Use locks for protecting updates, and use versions to detect conflicts involving reads

Global clock with per-object metadata (TL2)

Fixed global metadata (JudoSTM, RingSTM, NOrec STM)

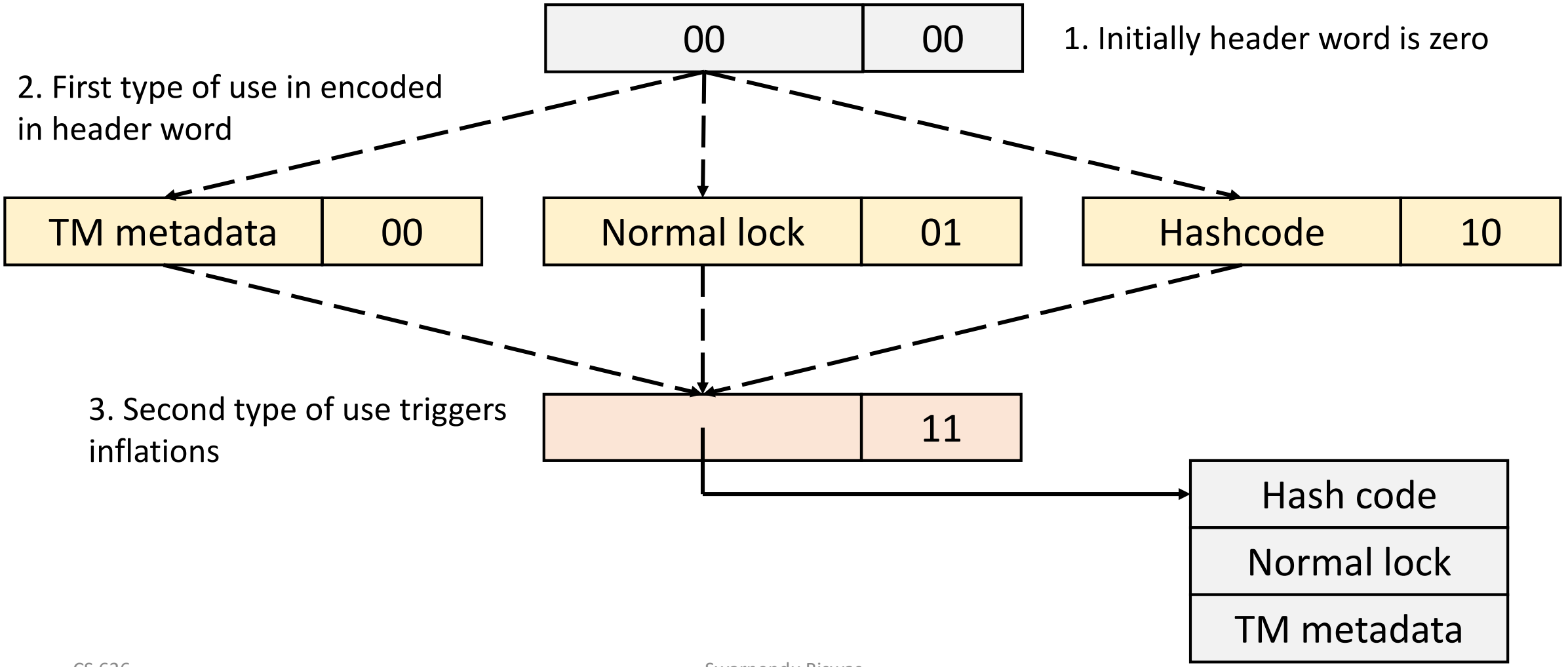
Nonblocking STMs (DSTM)

- Does not use locks

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 per-object version numbers

Header Word Optimizations in Bartok STM



Other Design Choices

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

Access-time locking
- Can support both eager or lazy version management
- Detects conflicts between active transactions, irrespective of whether they ultimately commit

Commit-time locking
- Can support only lazy version management

STM Metadata

Versioned locks

- Lock – mutual exclusion of 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
- Invisible reads – presence of a reading Tx is not visible to concurrent Txs which might try to commit updates to the objects being read

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);
}
```

Type specialization

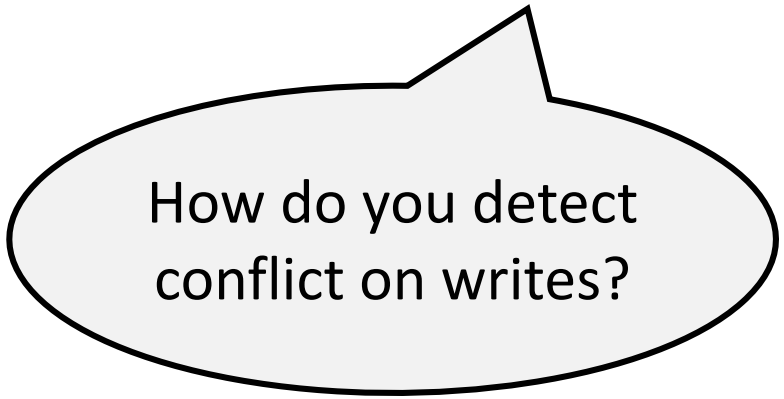
```
writeTx(tx, obj, off, newVal) {
  acquire(obj);
  undoLogInt(tx, obj, off);
  tx.writeSet.obj = obj;
  tx.writeSet.off = off;
  tx.writeSet.ver = ver;
  tx.writeSet++;
  write(obj, off, newVal);
  release(obj);
}

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

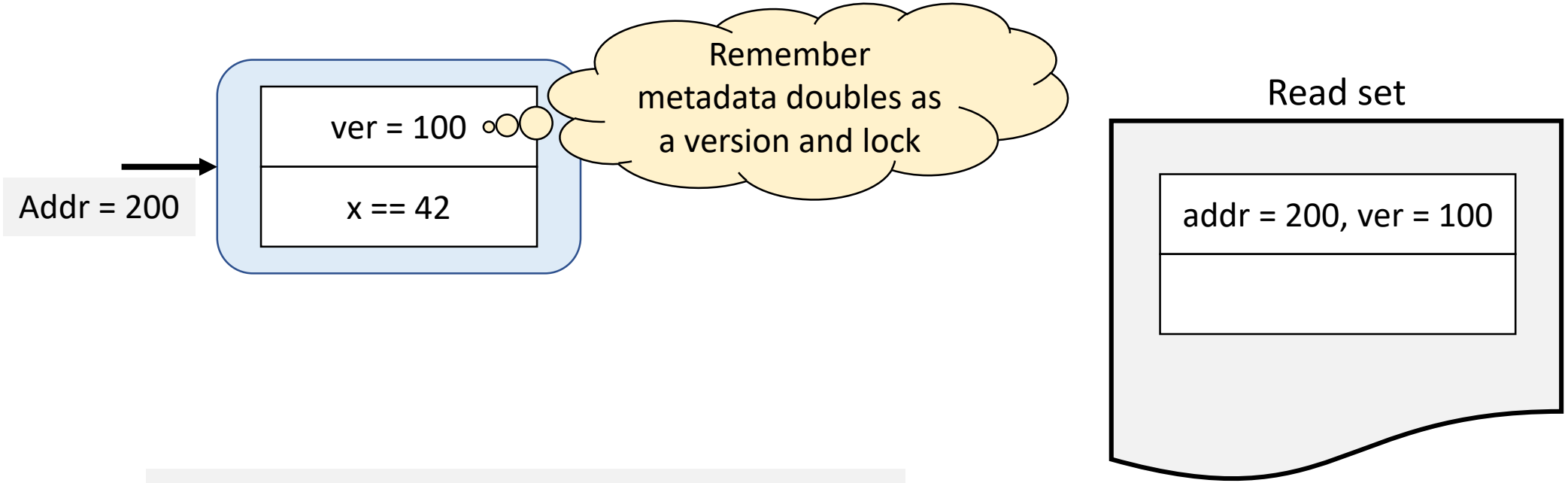
Writes

Reads?

Unlock increments the version number

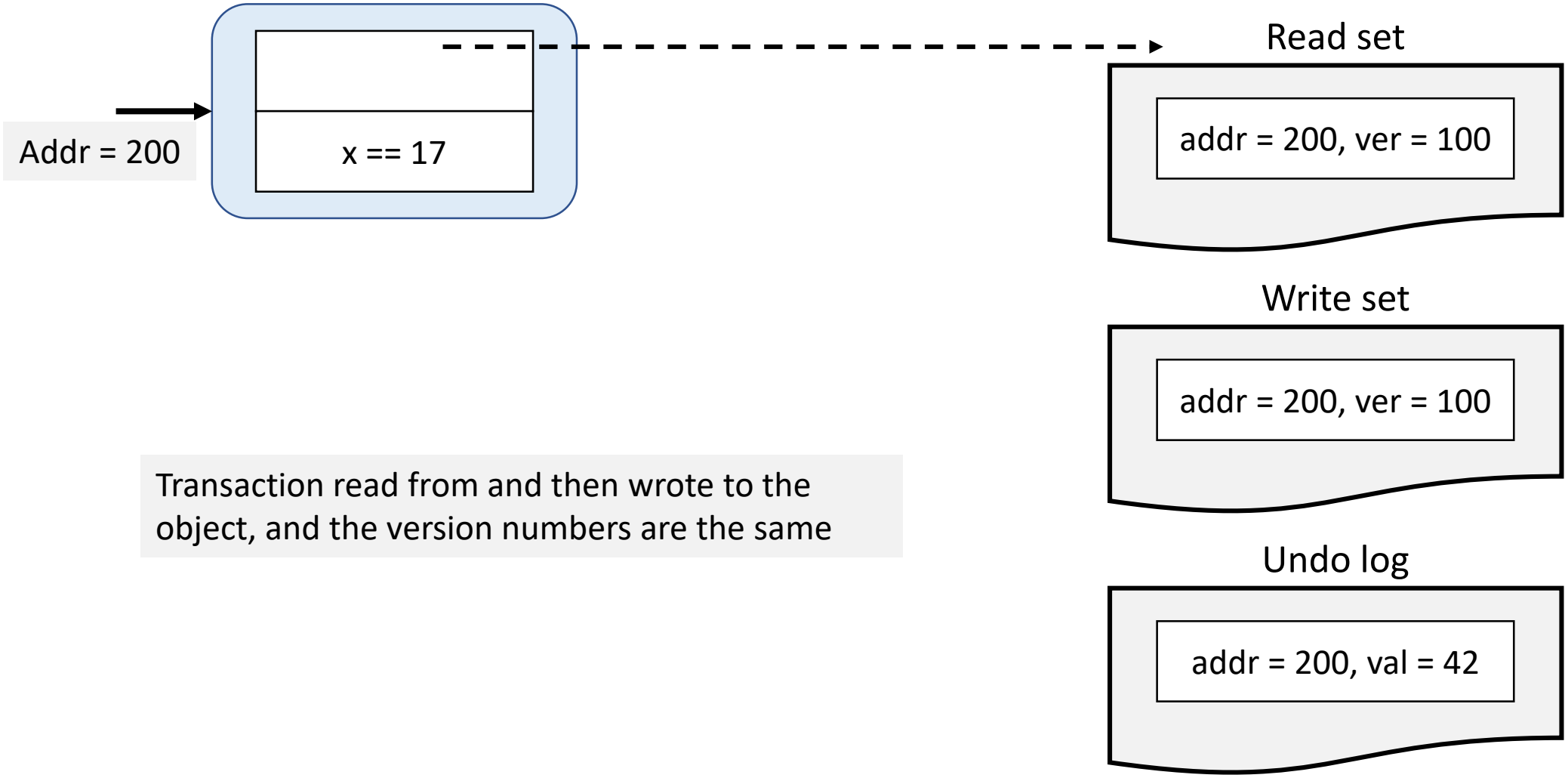
```
bool commitTx(tx) {
  foreach (entry e in tx.readSet)
    if (!validateTx(e.obj, e.ver))
      abortTx(tx);
      return false;
  foreach (entr e in tx.writeSet)
    unlock(e.obj, e.ver);
  return true;
}
```

No Conflict on Read from Addr=200



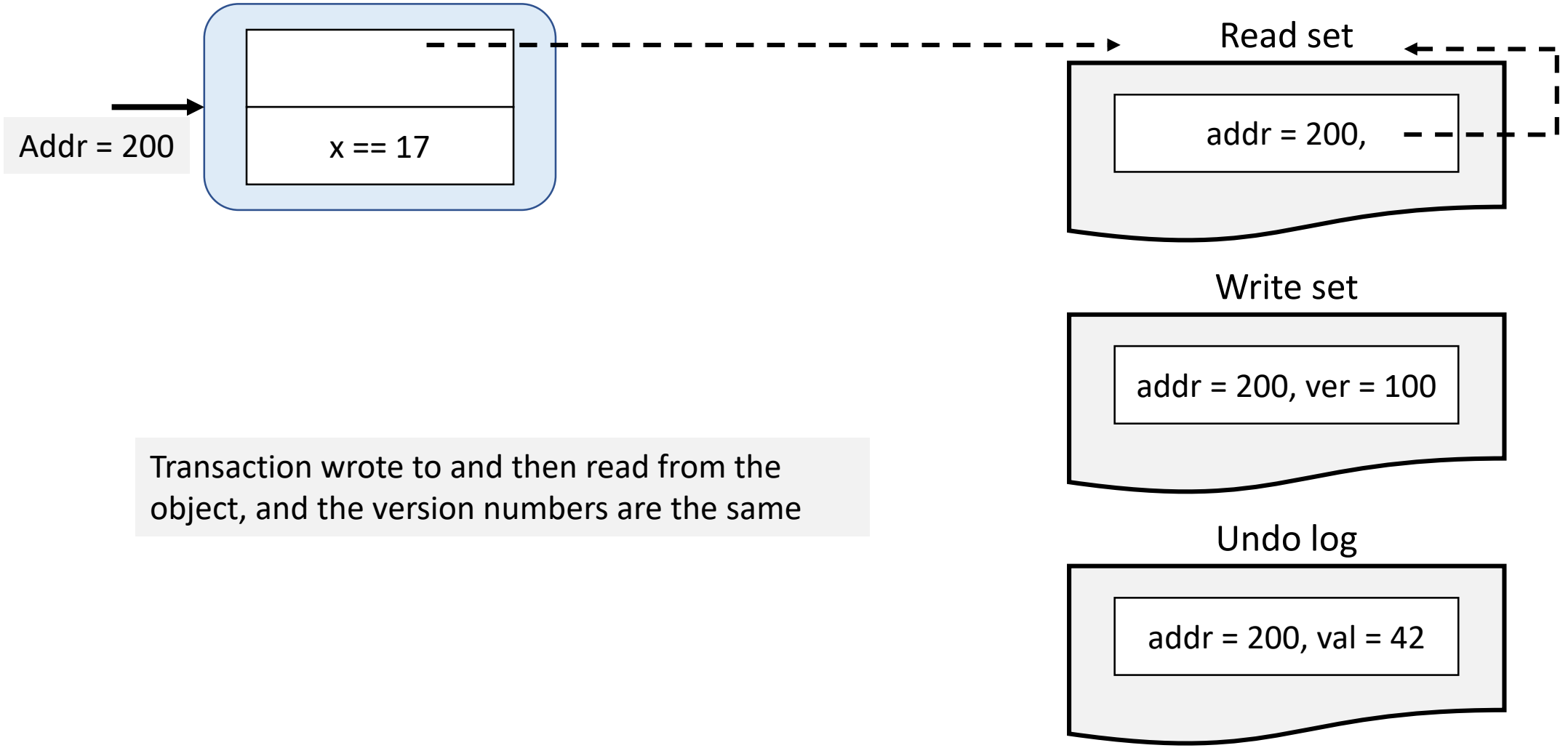
Transaction read from the object, and its version number is unchanged at commit time

No Conflict on Read from and Write to Addr=200



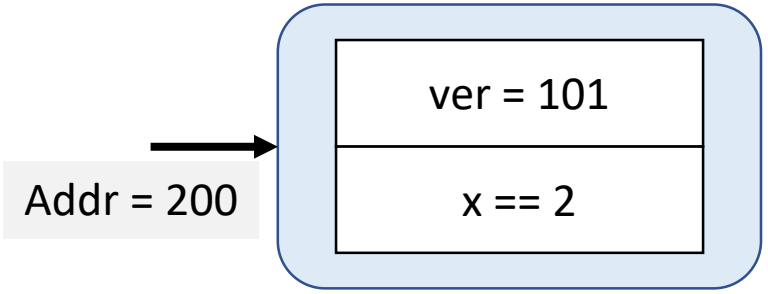
Transaction read from and then wrote to the object, and the version numbers are the same

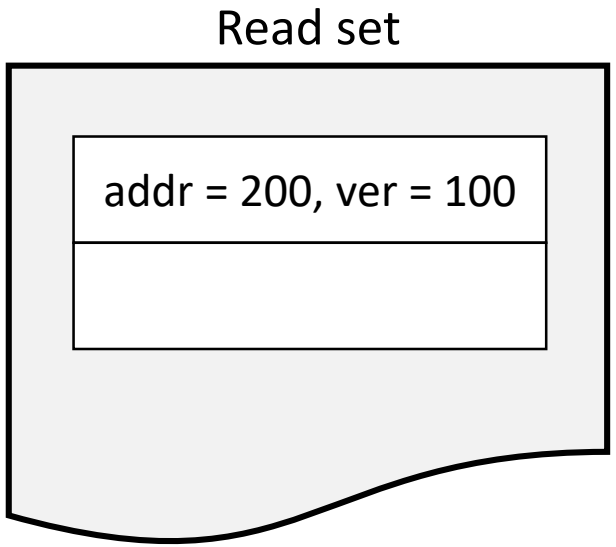
No Conflict on Write to and Read from Addr=200



Transaction wrote to and then read from the object, and the version numbers are the same

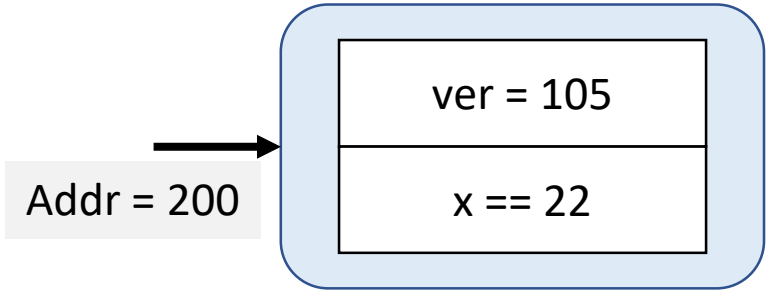
Conflict on Read from Addr=200, Concurrent Tx Updates and Commits

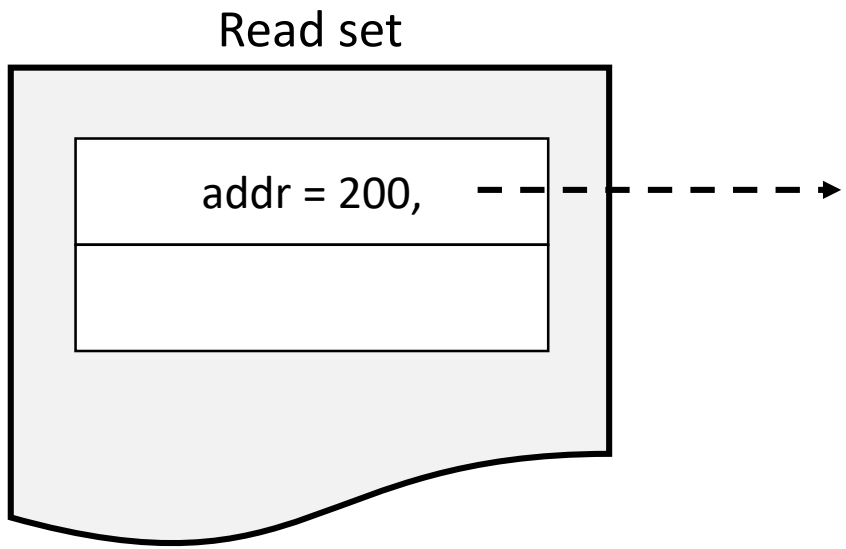




Transaction read from the object, and there is a version mismatch during `commitTx()`

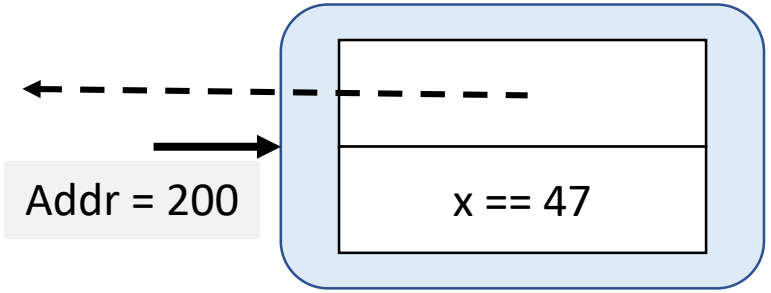
Conflict on Read from Addr=200, Concurrent Write

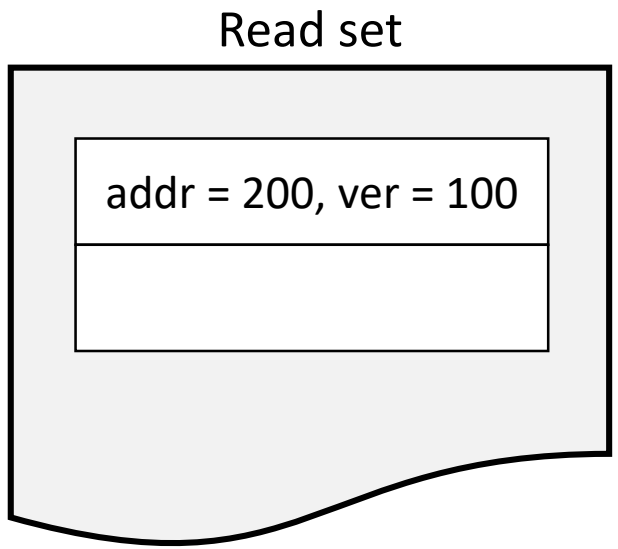




Transaction read from the object when it was owned by some other Tx

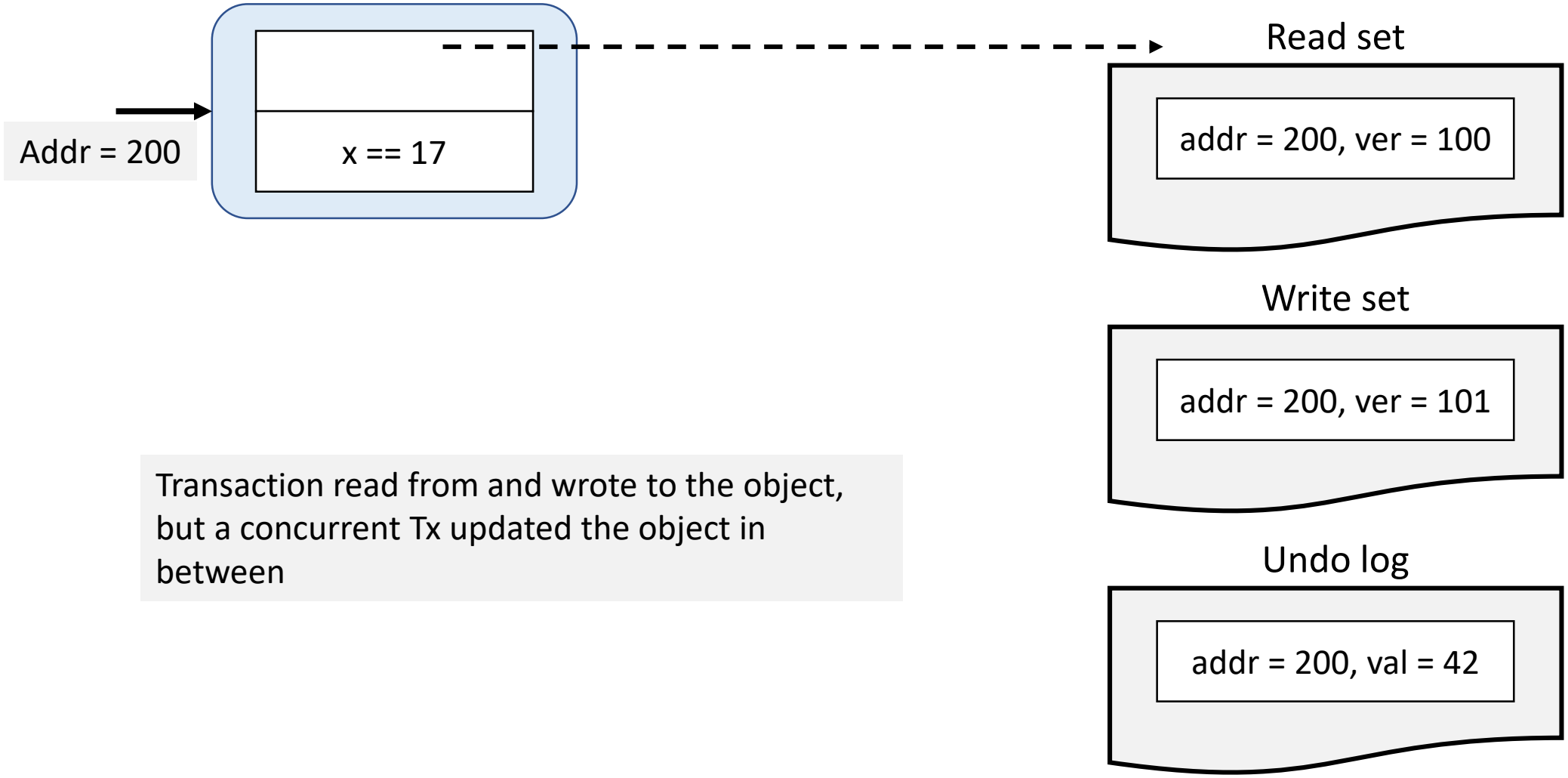
Conflict on Read from Addr=200 during Commit





Transaction is owned by some other Tx when the current reader Tx tries to commit

Conflict Between Read and Write from Addr=200



Transaction read from and wrote to the object, but a concurrent Tx updated the object in between

Practical Issues

Version overflow

- Theoretical concern, is a practical concern if the metadata is “packed”
- Globally renumber objects if overflow is rare
- Distinguish between an “old” and a wrapped-around “new” version
  - Ensure that each thread validates its current Tx at least once within n version increments

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

Swarnendu Biswas

Hardware Transactional Memory (HTM)

- Can provide strong isolation without modifications to non-Tx accesses
- Easy to extend to unmanaged languages

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

Possible ISA Extensions

Similar to STMs, HTMs need to demarcate Tx boundaries and transactional memory accesses

Explicit

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

Memory accessed within a Tx through ordinary memory instructions do not participate in any transactional memory protocol

Implicit

- `begin_transaction`
- `end_transaction`

All memory accesses are transactional

Which is simpler?

Comparison

Explicitly Transactional HTMs	Implicitly Transactional HTMs

• Provides flexibility to choose desired memory locations
  • Reduced read and write set size
• May require multiple library versions
  • Limits reuse of legacy libraries in HTMs
	• Larger read and write sets
	• Easy to reuse software libraries

Design Issues in HTMs

need to be careful with writes

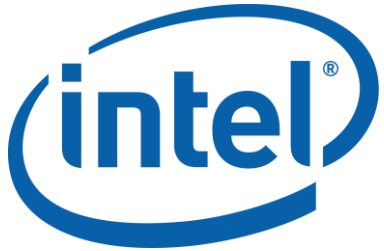
Tracking read and write sets

- Introducing additional structures like transactional cache complicates the data path
- Recent ideas extend existing data caches to track accesses
  - Granularity matters (one read bit for a cache line)

Conflict detection

- Natural to piggyback on cache coherence protocols to detect conflicts
- Most HTMs detect conflicts eagerly, and transfer control to a software handler

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

High-Level Goal with Transactions

- Hardware dynamically determines whether threads need to serialize
  - For example, with lock-protected critical sections
- Hardware serializes only when required
- Thus, processor exposes and exploits concurrency that is hidden due to unnecessary synchronization
- Lock elision idea introduced by Ravi Rajwar and James R. Goodman in 2001
  - Remove locks, run code as a transaction
  - If there are conflicts, abort and rerun code with locks intact
  - On success, commit the transaction's writes to memory

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`

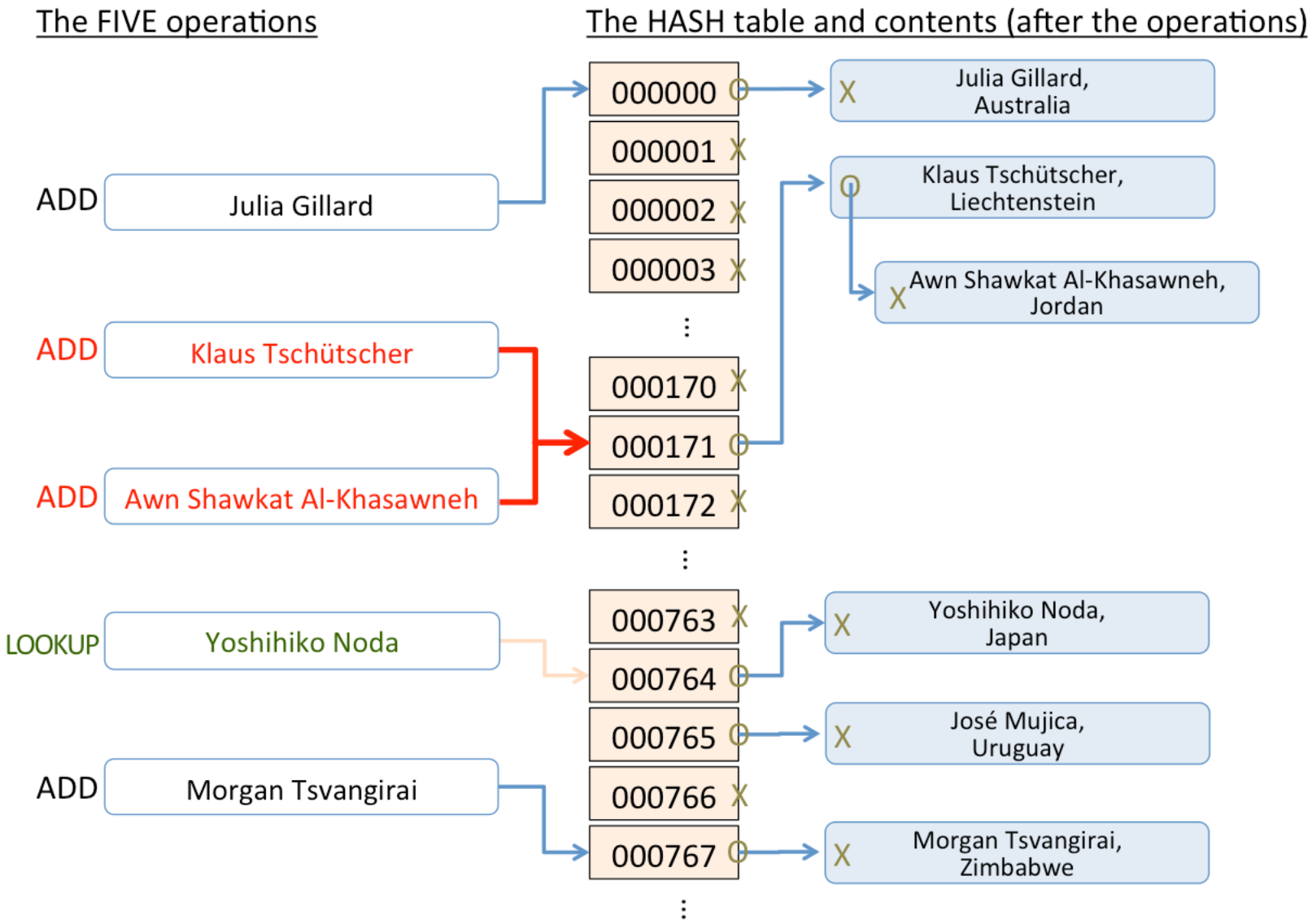
- New ISA extensions

Hardware Lock Elision (HLE)

- Application uses legacy-compatible prefix 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.

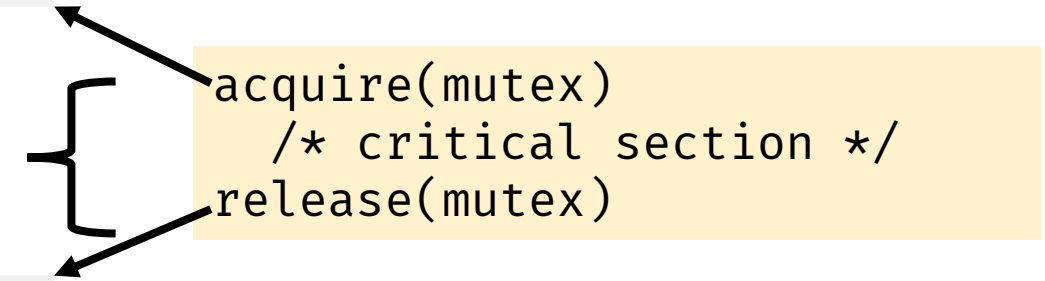
Goal with Intel TSX



https://software.intel.com/content/dam/develop/external/us/en/images/slide1.png

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
```




```
        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
```

application

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

```
        mov mutex, 0
```

```
        xrelease mov mutex, 0
```

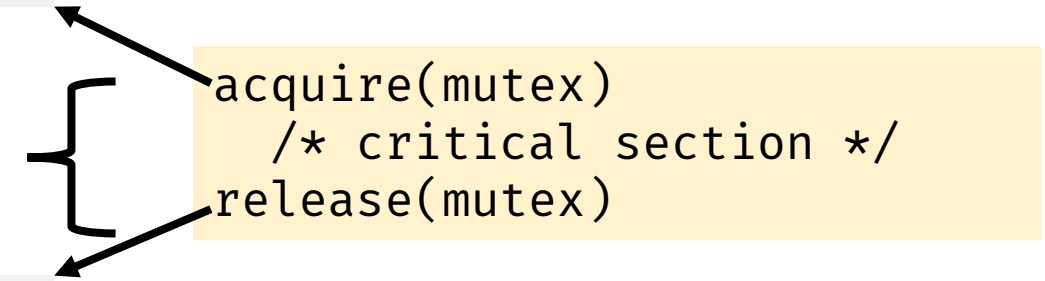
Intel Transactional Synchronization Extensions. Intel Developer Forum 2012.

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
- Processor may abort RTM transactional execution for several reasons
- 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
```




```
        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
       …
```

acquire (mutex)

```
        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
```

release (mutex)

Intel Transactional Synchronization Extensions. Intel Developer Forum 2012.

XTEST

- XTEST instruction
  - Queries whether the logical processor is transactionally executing in a transactional region identified by either HLE or RTM

Aborts in TSX

- Conflicting accesses from different cores (data, locks, false sharing)
  - TSX maintains read/write sets at the granularity of cache lines
- 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

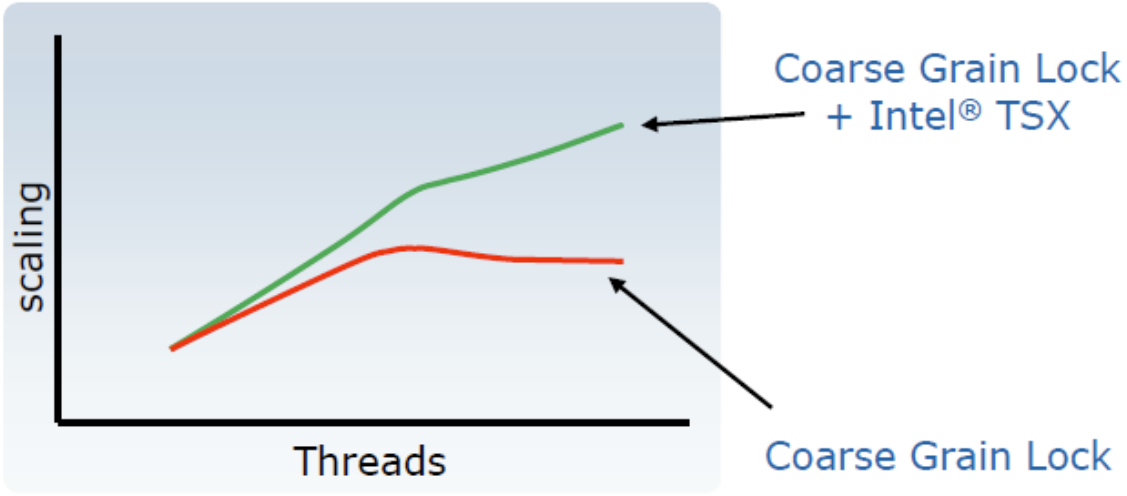
- Every detail is not known
  - Read and write sets are at cache line granularity
  - Uses L1 data cache as the storage
- Conflict detection is through cache coherence protocol

TSX Caveats

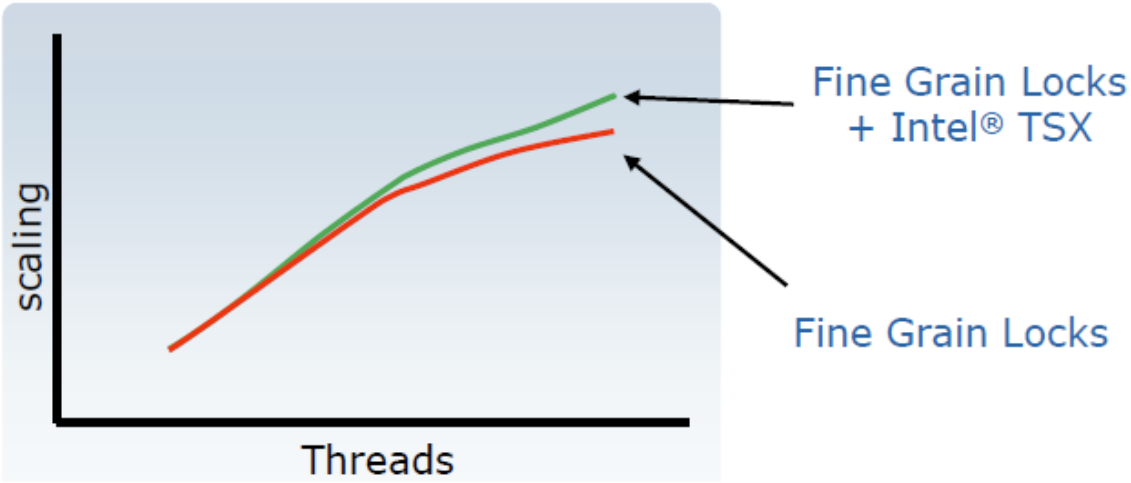
- No guarantees that Txs will commit
- There should be a software fallback independent of TSX to guarantee forward progress

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.
- 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
- Intel Architecture Instruction Set Extensions Programming Reference. Sections 8.1—8.2.
- Ravi Rajwar and Martin Dixon. Intel Transactional Synchronization Extensions. Intel Developer Forum 2012.