

CS 636: Transactional Memory

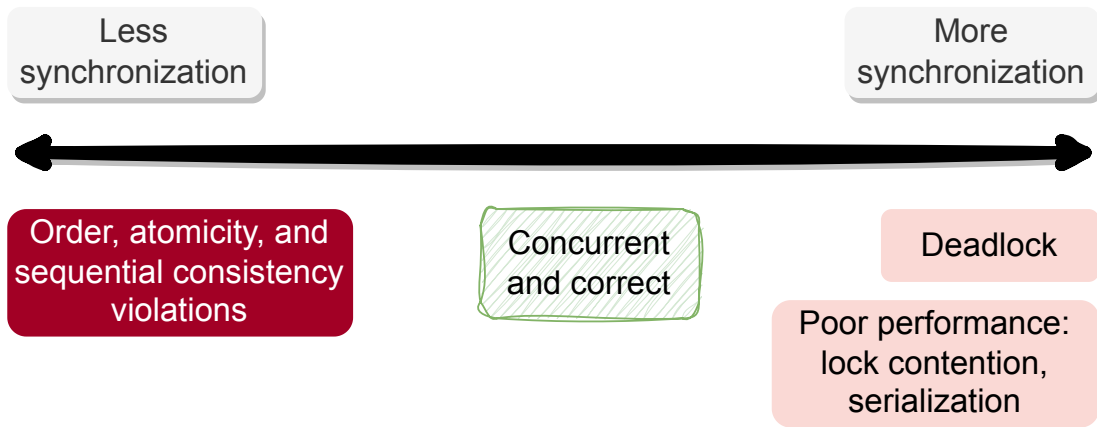
Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

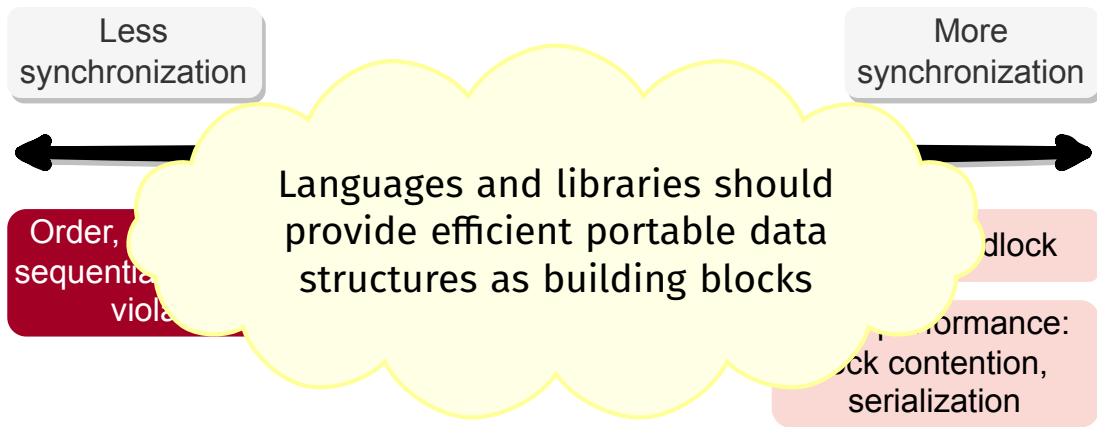
Sem 2025-26-II



Challenges with Concurrent Programming

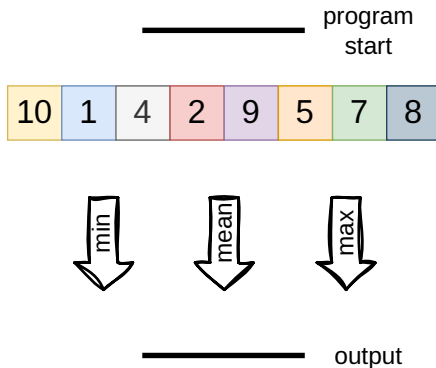


Challenges with Concurrent Programming



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

```
1 public Object get(Object key) {  
2     int idx = hash(key); // Compute hash to find bucket  
3     HashEntry e = buckets[idx];  
4     while (e != null) { // Find element in bucket  
5         if (equals(key, e.key))  
6             return e.value;  
7         e = e.next;  
8     }  
9     return null;  
10 }
```

No lock overhead but
thread-unsafe

Synchronized HashMap in Java

```
1 public Object get(Object key) {  
2     synchronized (mutex) { // mutex guards all accesses  
3         return myHashMap.get(key);  
4     }  
5 }
```

Thread-safe but uses explicit coarse-grained locking

Coarse-Grained and Fine-Grained Locking

Coarse-Grained Locking

- + Easy to implement correctly
- Limits concurrency, poor scalability

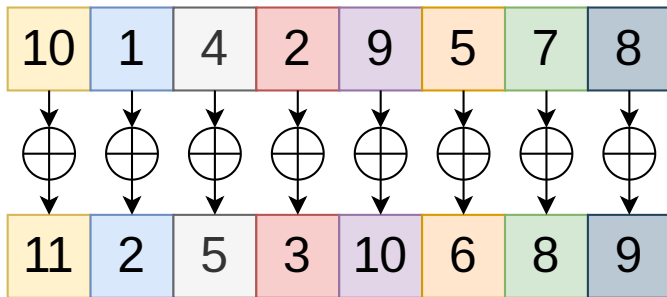
Fine-Grained Locking

- + More concurrency, better performance
- Difficult to get correct, more error-prone

Data Parallelism

Same task is applied on many data items in parallel

- E.g., processing pixels in an image
- Useful for numeric computations
- Not a universal programming model



Task vs Data Parallelism

Task Parallelism

- Different operations on same or different data
- Generic programming model that can express all forms of parallel computation
- Parallelization depends on task decomposition
- Speedup is less because of limited parallelization opportunities and synchronization

Data Parallelism

- Same operation on different data
- Not a generic programming model
- Parallelization proportional to the input data size
- Speedup is usually more

Abstraction and Composability

Programming models provide abstraction and composition

- For example, procedures, ADTs, and libraries
- Abstraction is a simplified view of an entity or a problem (e.g., procedures and ADT)
- Composability joins smaller units to form larger, more complex units (e.g., library methods)

Abstraction and Composability

Programming models provide abstraction and composition

- For example, procedures, ADTs, and libraries
- Abstraction is a simplified view of an entity or a problem (e.g., procedures and ADT)
- Composability joins smaller units to form larger, more complex units (e.g., library methods)

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 wakeup — missed notify for condition variable
- Deadlock
- Priority inversion
- Lock convoying
- Locking relies on programmer conventions

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 wakeup — missed notify for condition variable
- Deadlock

— Prior

```
1 /*  
2  * When a locked buffer is visible to the I/O layer  
3  * BH_Launder is set. This means before unlocking  
4  * we must clear BH_Launder, mb() on alpha and then  
5  * clear BH_Lock, so no reader can see BH_Launder set  
6  * on an unlocked buffer and then risk to deadlock.  
7  */  
8
```

— Lock

— Lock

— Bradley Kuszmaul

Lock-based Synchronization is not Composable

```
1 class HashTable {  
2     void synchronized insert(T elem);  
3     boolean synchronized remove(T elem);  
4 }
```

You may now want to add a new method move()

```
1 boolean move(HashTable tab1, HashTable tab2, T elem)  
2     ⇒ remove()  
3     ⇒ insert()
```

Lock-based Synchronization is not Composable

```
1 class HashTable {  
2     void synchronized insert(T elem);  
3     boolean synchronized remove(T elem);  
4 }
```

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 deadlock

```
1 boolean move(HashTable tab1, HashTable tab2, T elem) {  
2     atomic {  
3         boolean res = tab1.remove(elem);  
4         if (res)  
5             tab2.insert(elem);  
6     }  
7     return res;  
8 }
```

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 has to say how work should be done with imperative abstraction

Easy programmability (like coarse-grained locks)

Goal is to achieve performance 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

💡 ACID properties

Atomicity whole transaction must succeed

Consistency invariant on the data values

Isolation no interference from concurrent transactions

Durability results persist

TM vs Database Transactions

TM

- Supported by language runtime or hardware
- Not durable
- Memory operations can be a mix of transactional and non-transactional accesses
- Operations are from main memory, performance is critical

Databases

- Application level concept
- Durable
- All accesses are transactional
- Operations involve mostly disk accesses

Properties of TM execution

Atomic Appears to happen instantaneously

Commit Updates appear atomic

Abort Has no side effects

Serializable Appears to happen serially in order

Isolation Other transactions cannot observe writes before commit

TM Execution Semantics

Thread 1

```
1 atomic {  
2   a = a - 20;  
3   b = b + 20;  
4   c = a + b;  
5   a = a - b;  
6 }
```

Thread 2

```
1 atomic {  
2   c = c + 40;  
3   d = a + b + c;  
4 }  
5  
6
```

TM Execution Semantics

Thread 1

```
1 atomic {  
2   a = a - 20;  
3   b = b + 20;  
4   c = a + b;  
5   a = a - b;  
6 }
```

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

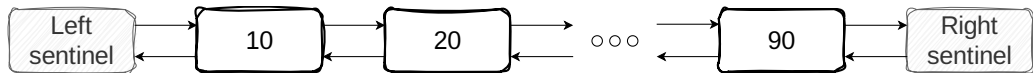
Thread 2

```
1 atomic {  
2   c = c + 40;  
3   d = a + b + c;  
4   }  
5
```

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

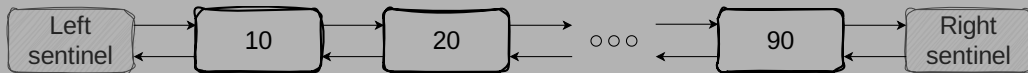
No data race due to TM semantics

Linked-List-based Double Ended Queue



```
1 void PushLeft(DQueue *q, int val) {  
2     QNode *qn = malloc(sizeof(QNode));  
3     qn->val = val;  
4     atomic {  
5         QNode *leftSentinel = q->left;  
6         QNode *oldLeftNode = leftSentinel->right;  
7         qn->left = leftSentinel;  
8         qn->right = oldLeftNode;  
9         leftSentinel->right = qn;  
10        oldLeftNode->left = qn;  
11    }  
12 }
```

Linked-List-based Double Ended Queue



```
1 void PushLeft(DQueue *q, int val) {  
2     QNode *qn = malloc(sizeof(QNode));  
3     qn->val = val;  
4     ...  
5 }
```

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
- Need to take care of corner cases (for example, only one element is left)

```
11 }
```

```
12 }
```

Atomicity Violations

Thread 1

```
1  ...
2  if (thd->proc_info) {
3
4
5
6      puts(thd->proc_info, ...)
7  }
8  ...
```

Thread 2

```
1
2  ...
3
4      thd->proc_info = NULL;
5
6  ...
7
8
```

MySQL: ha_innodb.cc

Fixing Atomicity Violations with TM

Thread 1

```
1 ...
2 atomic {
3     if (thd->proc_info) {
4         puts(thd->proc_info, ...)
5     }
6 }
7 ...
```

Thread 2

```
1
2
3
4
5
6 ...
7 atomic {
8     thd->proc_info = NULL;
9 }
10 ...
```

Fixing Atomicity Violations with TM

Thread 1

```
1  
2  
3  
4 ...  
5 atomic {  
6     if (thd->proc_info) {  
7         puts(thd->proc_info, ...)  
8     }  
9 }  
10 ...
```

Thread 2

```
1 ...  
2 atomic {  
3     thd->proc_info = NULL;  
4 }  
5 ...  
6  
7  
8  
9  
10
```

TM vs synchronized in Java

TM

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

synchronized

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

TM Interface

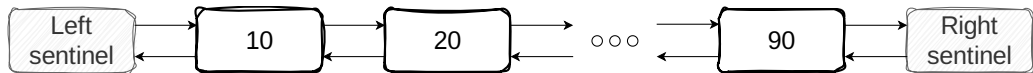
```
1 void startTx();  
2 bool commitTx();  
3 void abortTx();  
4 T readTx(T *addr);  
5 void writeTx(T *addr, T val);
```

Read set Set of variables read by the Tx

Write set Set of variables written by the Tx

Functions can be overloaded by types, or we can use generics

Linked-List-based Double Ended Queue with TM



```
1 void PushLeft(DQueue *q, int val) {  
2     QNode *qn = malloc(sizeof(QNode));  
3     qn->val = val;  
4     do {  
5         startTx();  
6         QNode *leftSentinel = readTx(&(q->left));  
7         QNode *oldLeftNode = readTx(&(leftSentinel->right));  
8         writeTx(&(qn->left), leftSentinel);  
9         writeTx(&(qn->right), oldLeftNode);  
10        writeTx(&(leftSentinel->right), qn);  
11        writeTx(&(oldLeftNode->left), qn);  
12    } while(!commitTx());  
13 }
```

- Similar to sequential code
- No explicit locks

Transactions cannot replace all uses of locks

Thread 1

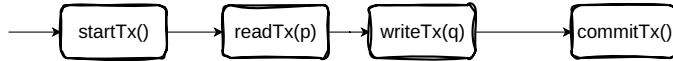
```
1 do {  
2   startTx();  
3   writeTx(&x, 1);  
4 } while (!commitTx());  
5
```

Thread 2

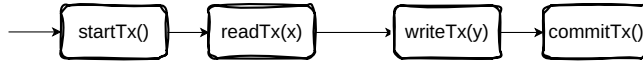
```
1 do {  
2   startTx();  
3   int tmp = readTx(&x);  
4   while (tmp == 0) {}  
5 } while (!commitTx());
```

Concurrency in TM

Thread 1

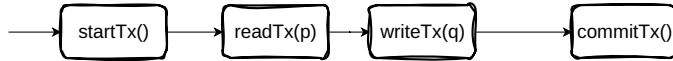


Thread 2

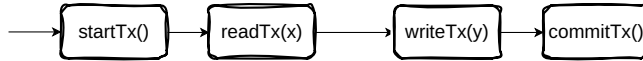


Concurrency in TM

Thread 1



Thread 2



Two levels

- (i) Among Tx's from concurrent thread
- (ii) Among individual Tx operations

Design Choices

Concurrency Control, Version Management, and 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 Txs i and j . Then, a conflict occurs iff
 - $R_i \cap W_j \neq \emptyset$, or
 - $R_j \cap W_i \neq \emptyset$, or
 - $W_i \cap W_j \neq \emptyset$,
- 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
 - ▶ For example, 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;

Thread 1

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp + 100;  
4 }
```

Location	Value read	Value written
----------	------------	---------------

Thread 2

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp - 100;  
4 }
```

Location	Value read	Value written
----------	------------	---------------

TM Example Execution

bal = 1000;

Thread 1

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp + 100;  
4 }
```

Location	Value read	Value written
----------	------------	---------------

Thread 2

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp - 100;  
4 }
```

Location	Value read	Value written
----------	------------	---------------

bal	1000	
-----	------	--

TM Example Execution

bal = 1000;

Thread 1

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp + 100;  
4 }
```

Location	Value read	Value written
----------	------------	---------------

bal	1000	
-----	------	--

Thread 2

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp - 100;  
4 }
```

Location	Value read	Value written
----------	------------	---------------

bal	1000	
-----	------	--

TM Example Execution

bal = 1000;

Thread 1

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp + 100;  
4 }
```

Location	Value read	Value written
----------	------------	---------------

bal	1000	1100
-----	------	------

Thread 2

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp - 100;  
4 }
```

Location	Value read	Value written
----------	------------	---------------

bal	1000	
-----	------	--

TM Example Execution

bal = 1100;

Thread 1

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp + 100;  
4 }
```

Location	Value read	Value written
bal	1000 ✓	1100

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

Thread 2

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp - 100;  
4 }
```

Location	Value read	Value written
bal	1000	

TM Example Execution

bal = 1100;

Thread 1

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp + 100;  
4 }
```

Thread 2

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp - 100;  
4 }
```

Location	Value read	Value written
bal	1000	900

TM Example Execution

bal = 1100;

Thread 1

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp + 100;  
4 }
```

Thread 2

```
1 atomic {  
2   tmp = bal;  
3   bal = tmp - 100;  
4 }
```

Location	Value read	Value written
bal	1000 X	900

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

Concurrency Control

i Pessimistic

- Occurrence of a conflict, and its detection and resolution happen at the **same time** during execution
- Claims ownership of data before modifications, locking out other interested transactions

i Optimistic

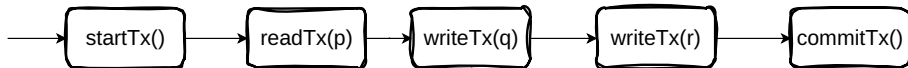
- Conflict detection and resolution can happen **after** the conflict occurs
- Multiple conflicting transactions can continue to keep running, as long as the conflicts are detected and resolved before the Tx commit

Time of locking

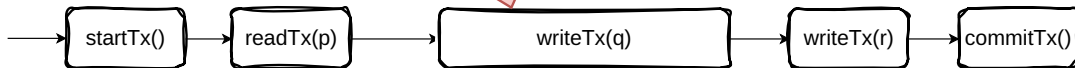
- When the Tx first accesses a location (i.e., access-time locking)
- When the Tx is about to commit (i.e., commit-time locking)

Pessimistic Concurrency Control

Thread 1

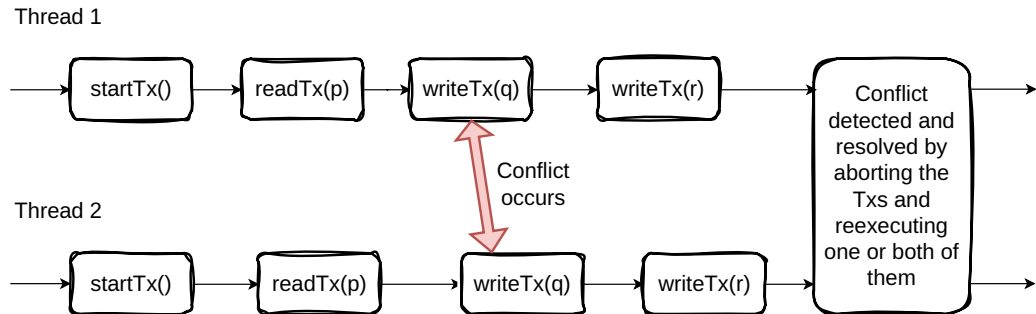


Thread 2



Conflict occurs, is detected, and is resolved by delaying Thread 2's Tx

Optimistic Concurrency Control



Pessimistic vs Optimistic Concurrency Control

Pessimistic

- Usually claims exclusive ownership of data before accessing
- Needs to avoid or detect and recover from deadlock situations
- + Effective in high contention cases

Optimistic

- Avoids claiming exclusive ownership of data, provides more conflict resolution choices
- Needs to avoid livelock situations through contention management schemes
- + Effective in low contention cases

Hybrid Concurrency Control

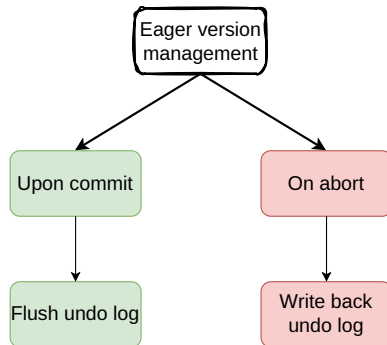
- Use pessimistic control for writes and optimistic control for reads
- Use optimistic control TM with pessimistic control of **irrevocable** Tx
 - ▶ Irrevocable Tx means that the changes cannot be rolled back
 - ▶ A Tx that has performed I/O or a Tx that has experienced frequent conflicts in the past are irrevocable Tx

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

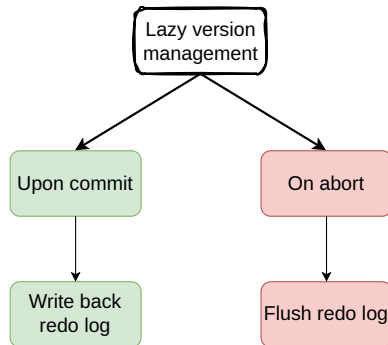


Version Management

TMs need to track updates for conflict resolution

- Lazy

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



Understanding Conflict Detection

- Pessimistic concurrency control is straightforward

Understanding Conflict Detection

- Pessimistic concurrency control is straightforward
- Which concurrency control type should we use with eager version management, pessimistic or optimistic?

Understanding Conflict Detection

- Pessimistic concurrency control is straightforward
- Which concurrency control type should we use with eager version management, pessimistic or optimistic?
- How do you check for conflicts in optimistic concurrency control?

Conflict Detection in Optimistic Concurrency Control

💡 Conflict granularity

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

💡 Time of conflict detection

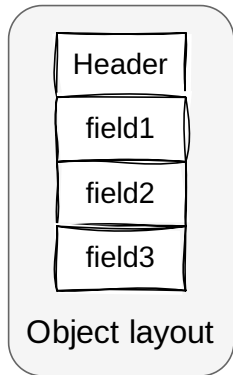
- Just before access (eager) or during validation before commit (lazy)
- Validation can occur at any time, and can occur multiple times during a transaction's execution

💡 Types of conflicting accesses

Among concurrent ongoing Txns using eager mechanisms, or between active and committed Txns using lazy mechanisms

Object Layout

Object Model in Jikes RVM



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

  SCALAR LAYOUT:
  |<----- scalar header ----->|
  +-----+-----+-----+-----+-----+-----+-----+
  | GCHdr | MiscHdr | JavaHdr | fld0 | fld1 | fldx | fldN-1 |
  +-----+-----+-----+-----+-----+-----+-----+
                                     ^ JHOFF                ^objref
                                     .
                                     .

  ARRAY LAYOUT:
  |<----- array header ----->|
  +-----+-----+-----+-----+-----+-----+-----+
  | GCHdr | MiscHdr | JavaHdr | len | elt0 | elt1 | ... | eltN-1 |
  +-----+-----+-----+-----+-----+-----+-----+
                                     ^ JHOFF                ^objref
```

Challenges with Coarse-Grained Conflict Granularity

- x and y are two 1 B variables in the same 4 B word
- TM system uses a 4 B+ granularity for conflict detection

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

Thread 1

```
1 do {  
2   startTx();  
3   tmp = readTx(&x);  
4   writeTx(x, 10);  
5 } while (!commitTx());
```

Thread 2

```
1 ...  
2 y = 20;  
3 ...  
4  
5
```

Weak atomicity/isolation can lead to granular lost update (GLU)

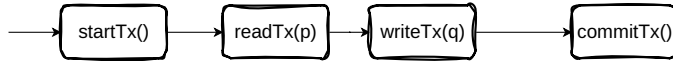
Transaction Semantics

Concurrency in TM

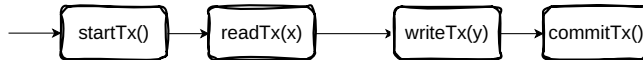
Two levels

- (i) Among Txns from concurrent thread
- (ii) Among individual Tx operations

Thread 1



Thread 2

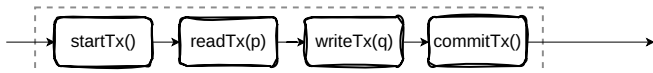


Serializability

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

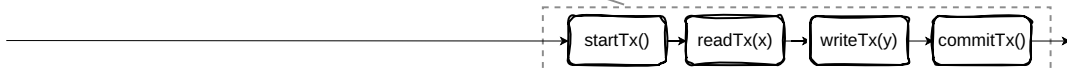
- Widely-used correctness condition in databases
- The TM system can reorder transactions
- Serializability requires the Txs **appear** to run in serial order
 - ▶ Does not require that the order has to be real-time

Thread 1



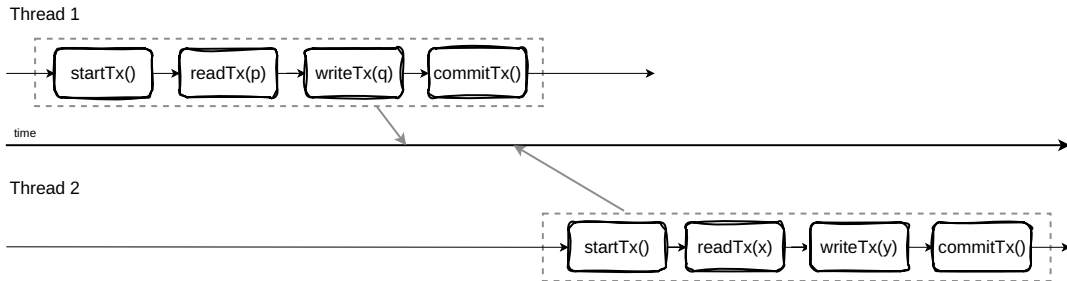
time

Thread 2

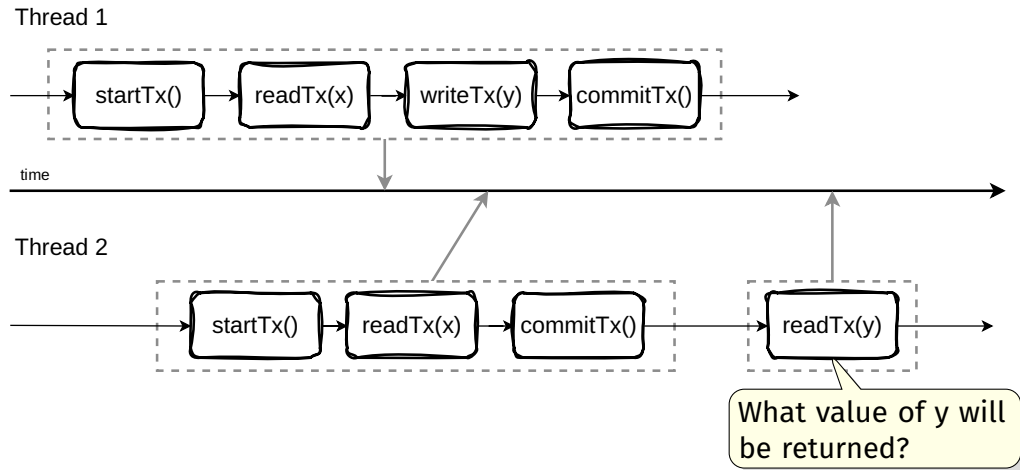


Strict Serializability

In strict serializability, if transaction TA completes before transaction TB starts, then TA must occur before TB in the equivalent serial execution



Limitations of Strict Serializability



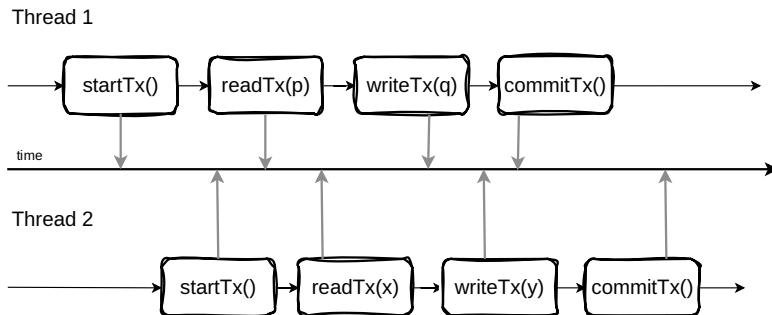
Definitions

- 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

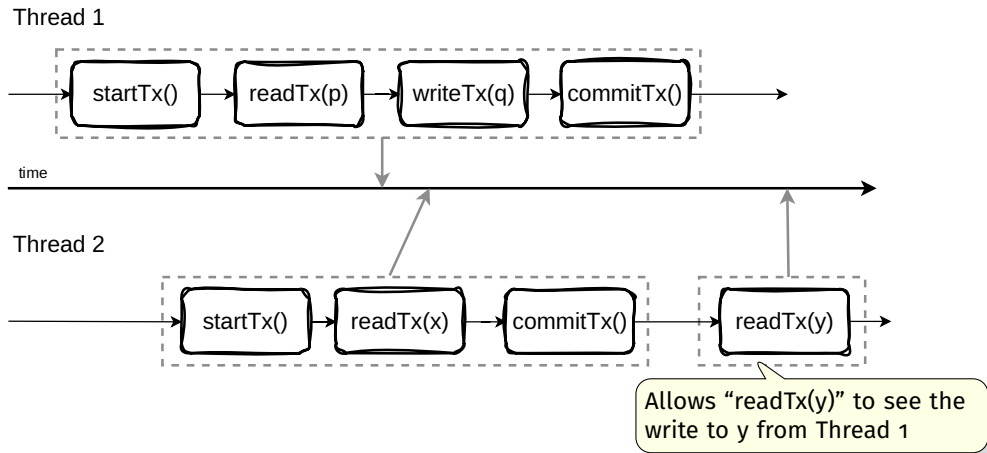
Linearizability of an operation (e.g., method call)

- (i) Method calls should appear to happen one-at-a-time in sequential order
- (ii) Each operation appears to execute atomically at some point between its invocation and its completion



Linearizability

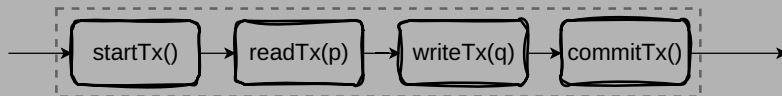
Linearizability of a transaction: a transaction is a single operation extending from the beginning of `startTx()` until the completion of its final `commitTx()`



Linearizability

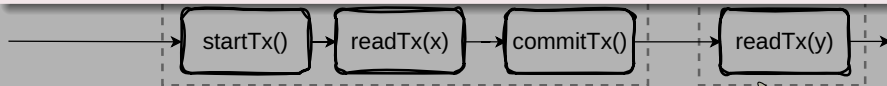
Linearizability of a transaction: a transaction is a single operation extending from the beginning of `startTx()` until the completion of its final `commitTx()`

Thread 1



time

If each transaction appears to execute atomically at a single instant, then conflicts between transactions will not occur



Allows “`readTx(y)`” to see the write to `y` from Thread 1

Snapshot Isolation (SI)

- SI is a weaker isolation requirement than serializability that 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
- Can potentially allow greater concurrency between Txs
- Many database implementations actually provide SI

SI allows concurrent transactions to see the same memory snapshot and then commit separate sets of updates that conflict with the snapshot but not with one another

Example of SI

```
1 x = 0;  
2 y = 0;
```

Thread 1

```
1 do {  
2   startTx();  
3   int tmp_x = readTx(x);  
4   int tmp_y = readTx(y);  
5   int tmp = tmp_x + tmp_y + 1;  
6   writeTx(x, tmp);  
7 } while (!commitTx());
```

Thread 2

```
1 do {  
2   startTx();  
3   int tmp_x = readTx(x);  
4   int tmp_y = readTx(y);  
5   int tmp = tmp_x + tmp_y + 1;  
6   writeTx(y, tmp);  
7 } while (!commitTx());
```

What are possible values of x and y after execution

(i) with serializability?

(ii) with SI?

Understanding SI

```
x = 0;
```

Thread 1

```
1 int t = x + 1; // 1
2 x = t;
```

Thread 2

```
1 int t = x + 1; // 1
2 x = t;
```

Sequentially consistent but not SI

Understanding SI

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

Thread 1

```
1 x = 1;  
2 int t = y; // 0
```

Thread 2

```
1 y = 1;  
2 int t = x; // 0
```

SI but not sequentially consistent
and not serializable

Understanding SI

- Semantics of SI may seem unexpected when compared with simpler models based on serial ordering of complete transactions
- Potential increase in 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 Txns

`x = y = 0;`

Assume eager version management
and lazy conflict detection

Thread 1

```
1 do {  
2   startTx();  
3   int tmp1 = readTx(&x);  
4  
5  
6  
7  
8  
9   int tmp2 = readTx(&y);  
10  while (tmp1 != tmp2) {}  
11 } while (!commitTx());
```

Thread 2

```
1  
2  
3  
4 do {  
5   startTx();  
6   writeTx(&x, 10);  
7   writeTx(&y, 10);  
8 } while (!commitTx());  
9  
10  
11
```

Validation only during commit is
insufficient for this TM design

Considerations with Zombie Tx

- A Tx that is inconsistent but is not yet detected is called a **zombie** Tx
- Careful handling of zombie Tx's are required, especially for unsafe languages
 - ▶ Inconsistent values can potentially be used in pointer arithmetic to access unwanted memory locations
- Possible workarounds: perform periodic validations
 - ▶ Validating n locations once requires n memory accesses, increases run-time overhead
 - ▶ 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 Tx
- Requires the non-Tx thread to participate in conflict detection

i Weak Atomicity (isolation)

Checks for conflicts and provides Tx semantics only among Tx

i Strong Atomicity (isolation)

Guarantees Tx semantics among Tx and non-Tx

Challenges with Weak Atomicity

Weak atomicity allows violating a transaction's isolation and consistency guarantees if there is a data race between transactional and non-transactional code

Programmers need to enclose all shared-memory accesses inside transactions to achieve isolation and consistency

Example with Lock-Based Synchronization

- Suppose variable `list` of type `java.util.LinkedList` is shared
- Initially `list == [Item{val1==0, val2==0}]`

Thread 1

```
1 Item item;  
2 synchronized(list) {  
3     item = list.removeFirst();  
4 }  
5 int r1 = item.val1;  
6 int r2 = item.val2;  
7
```

Thread 2

```
1 synchronized(list) {  
2     if (!list.isEmpty()) {  
3         Item item = list.getFirst();  
4         item.val1++;  
5         item.val2++;  
6     }  
7 }
```


Can we safely replace synchronize with atomic?

Consider a weakly atomic TM design with eager versioning and lazy conflict detection

- Suppose variable `list` of type `java.util.LinkedList` is shared
- Initially `list == [Item{val1==0, val2==0}]`

Thread 1

```
1 Item item;  
2 atomic(list) {  
3     item = list.removeFirst();  
4 }  
5 int r1 = item.val1;  
6 int r2 = item.val2;  
7
```

Thread 2

```
1 atomic(list) {  
2     if (!list.isEmpty()) {  
3         Item item = list.getFirst();  
4         item.val1++;  
5         item.val2++;  
6     }  
7 }
```

What can go wrong?

Few Issues to Consider with Weak Isolation

Shared with locks	Non-repeatable reads
	Intermediate lost updates
	Intermediate dirty reads
Eager versioning	Speculative lost updates
	Speculative dirty reads
Lazy versioning	Memory inconsistency
Coarse-grained versioning	Granular lost updates
	Granular inconsistent reads

Non-repeatable Reads

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

Thread 1

```
1 atomic {  
2   r1 = x;  
3  
4   r2 = x; // r1 != r2  
5 }
```

Thread 2

```
1  
2  
3 x = 1;  
4  
5
```

Unless the TM buffers the value seen by the first read, the transaction will see the update

Intermediate Lost Update

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.

```
x = 0;
```

Thread 1

```
1 atomic {  
2   r = x;  
3  
4   x = r + 1; // x == 1  
5 }
```

Thread 2

```
1  
2  
3 x = 10;  
4  
5
```

Intermediate Dirty Read

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.

```
assert (x%2 == 0);
```

Thread 1

```
1 atomic {  
2   X++;  
3  
4   X++;  
5 }
```

Thread 2

```
1  
2  
3 r = x;  
4  
5
```

Speculative Lost Updates

In speculative lost update, a non-transactional update is lost due to a write during transaction rollback

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

Thread 1

```
1 atomic {  
2   if (y == 0)  
3     x = 1;  
4  
5  
6   // abort  
7 }
```

Thread 2

```
1  
2  
3  
4 x = 2;  
5 y = 1;  
6  
7
```

Single-Lock Atomicity (SLA) for Transactions

How do we provide semantics for mixed-mode accesses?

With SLA, a program executes as if all transactions acquire a single, program-wide mutual exclusion lock

The example to highlight the limitation of strict serializability will work correctly with SLA

Thread 1

```
1 startTx();  
2 while (true) {}  
3 commitTx();
```

Thread 2

```
1 startTx();  
2 int tmp = readTx(&x);  
3 commitTx();
```

What will happen with (i) a global lock and (ii) SLA?

Nested Transactions

Execution of a nested Tx is wholly contained in the dynamic extent of another Tx

- Modifications made by the inner Tx becomes visible to the outer Tx on commit
- Modifications made by the inner Tx becomes visible to other threads only when the outermost Tx commits

```
1  int x = 1;
2  do {
3      startTx();
4      writeTx(&x, 2);
5      do {
6          startTx();
7          writeTx(&x, 3);
8          abortTx();
9          ...
```

Many choices on how nested Tx's interact

Flattened Aborting the inner Tx causes the outer Tx to abort

Closed Inner Tx can abort without terminating its parent Tx

- Open**
- Modifications made by an inner Tx becomes visible to all other Tx's even if the outer Tx is still running
 - The results of the nested Tx remain committed even if the parent Tx aborts

TM Implementations

Software Transactional Memory (STM)

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

Hardware Transactional Memory (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 (e.g., running or committed)
- Need to Maintain per-Tx read and write sets
 - ▶ Release locks acquired with pessimistic concurrency control
 - ▶ Allows detecting conflicts with optimistic concurrency control
- Maintain either redo log or undo log

- McRT-STM, PPOPP'06
- Bartok-STM, PLDI'06
- JudoSTM, PACT'07
- RingSTM, SPAA'08

- NoRec STM, PPOPP'10
- DeuceSTM, HiPEAC'10
- LarkTM, PPOPP'15
- ...

Implementing STM

- Use compilation passes to instrument the program

`startTx()` Tx entry point (prologue)

`commitTx()` Tx exit point (epilogue)

`readTx()` Tx read access

`writeTx()` Tx write access

- TM runtime tracks memory accesses, detects conflicts, and commits/aborts Tx

```
1 atomic {  
2     tmp = x;  
3     y = tmp + 1;  
4 }
```



```
1 // Per-TX data structure  
2 td = getTxDesc(thr);  
3  
4 startTx(td);  
5 tmp = readTx(&x);  
6 writeTx(&y, tmp+1);  
7 commitTx(td);
```

Importance of Undo and Redo Logs

Well-designed applications have low conflict rates

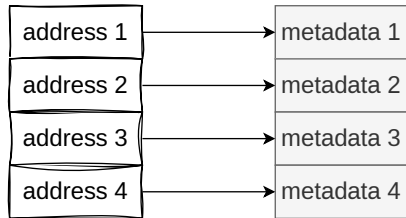
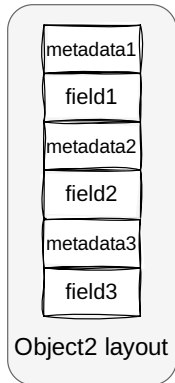
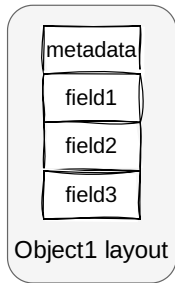
- Is the design of undo log important in a TM with eager version management?

Importance of Undo and Redo Logs

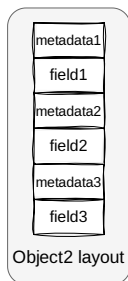
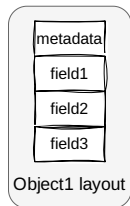
Well-designed applications have low conflict rates

- 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?

Object Metadata and Word Metadata



Pros and Cons of Metadata in Object Header



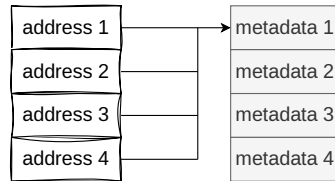
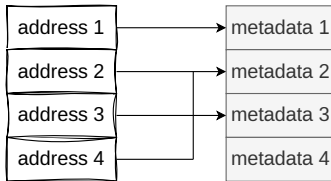
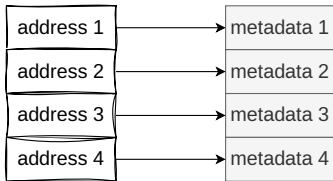
Pro

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

Con

- Potential for false conflicts
- Increases coupling (e.g., complicates GC)

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 among concurrent Txs

Impact on memory usage

Impact on performance, i.e., speed of mapping location to metadata

Major STM Designs

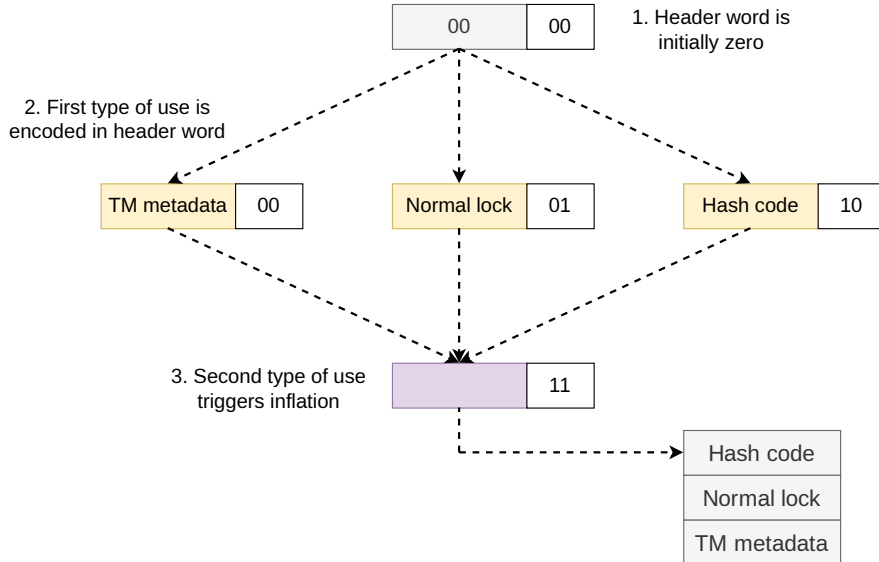
- (i) **Per-object** versioned locks (e.g., McRT-STM and Bartok-STM)
- (ii) **Global clock** with per-object metadata (e.g., TL2)
- (iii) **Fixed global** metadata (e.g., JudoSTM, RingSTM, and NOrec STM)
- (iv) **Nonblocking** STMs (e.g., DSTM) do not use locks

Lock-Based STM with Versioned Reads

High-level design

- Pessimistic concurrency control for writes
 - ▶ Locks are acquired dynamically for protecting updates
 - Optimistic concurrency control for reads
 - ▶ Validation using per-object version numbers to detect conflicts
- Design keeps the fast paths of reads of conflict-free transactions simple
 - ▶ Minimizes cache invalidations by not forcing readers to modify metadata
 - Version numbers are “local”—are incremented independently on an update

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

```
1 readTx(tx, obj, off) {  
2     tx.readSet.obj = obj;  
3     tx.readSet.ver = getVerFromMd(obj);  
4     tx.readSet++;  
5  
6     return read(obj, off);  
7 }
```

eager version
management

```
1 writeTx(tx, obj, off, newVal) {  
2     acquire(obj);  
3  
4     tx.undoLog.obj = obj;  
5     tx.undoLog.offset = off,  
6     tx.undoLog.val = read(obj, off);  
7     tx.undoLog++;  
8  
9     tx.writeSet.obj = obj;  
10    tx.writeSet.off = off;  
11    tx.writeSet.ver = obj.ver;  
12    tx.writeSet++;  
13  
14    write(obj, off, newVal);  
15    release(obj);  
16 }
```


Read and Write Operations

```
1 readTx(tx, obj, off) {  
2   tx.readSet.obj = obj;  
3   tx.readSet.ver = getVerFromMd(obj);  
4   tx.readSet++;  
5  
6   return read(obj, off);  
7 }
```

type specialization

```
1 writeTx(tx, obj, off, newVal) {  
2   acquire(obj);  
3   undoLogInt(tx, obj, off);  
4   tx.writeSet.obj = obj;  
5   tx.writeSet.off = off;  
6   tx.writeSet.ver = obj.ver;  
7   tx.writeSet++;  
8   write(obj, off, newVal);  
9   release(obj);  
10  
11  
12 undoLogInt(tx, obj, off) {  
13   tx.undoLog.obj = obj;  
14   tx.undoLog.offset = off,  
15   tx.undoLog.val = read(obj, off);  
16   tx.undoLog++;  
17 }
```

Conflict Detection on Writes

Writes

Reads

How do you detect conflicts on writes?

Conflict Detection on Reads

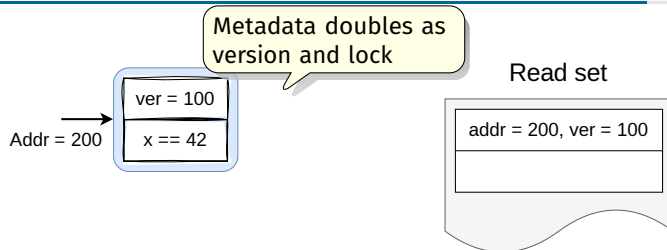
Writes

Reads

Unlock increments the version number

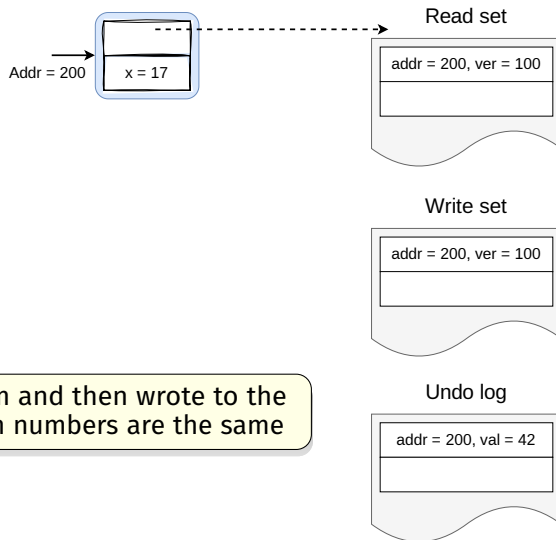
```
1 bool commitTx(tx) {  
2     foreach (entry e in tx.readSet)  
3         if (!validateTx(e.obj, e.ver)) {  
4             abortTx(tx);  
5             return false;  
6         }  
7  
8     foreach (entr e in tx.writeSet)  
9         unlock(e.obj, e.ver);  
10  
11     return true;  
12 }
```

No Conflict on Read from Addr=200



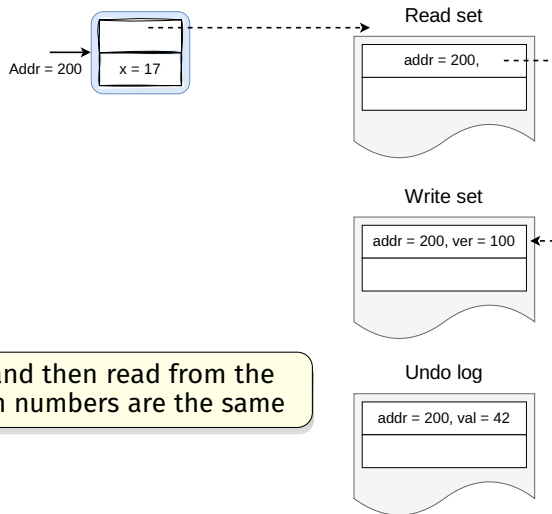
Transaction read from the object, and its version number is unchanged at commit time

No Conflict on Read from and Write to Addr=200



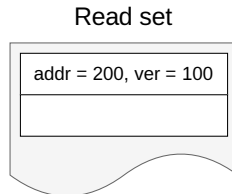
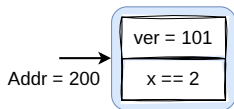
Transaction reads from and then wrote to the object, and the version numbers are the same

No Conflict on Write to and Read from Addr=200



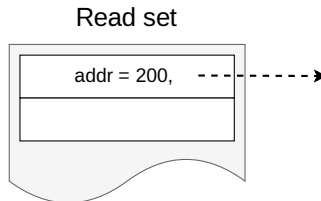
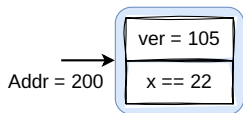
Transaction wrote to and then read from the object, and the version numbers are the same

Conflict on Read from Addr=200, Concurrent Tx Commits



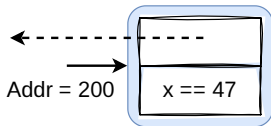
Transaction read from the object, and there is a version mismatch during `commitTx()`

Conflict on Read from Addr=200, Concurrent Write

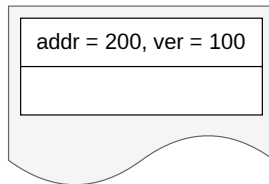


Transaction read from the object when it was owned by some other Tx

Conflict on Read from Addr=200 during Commit

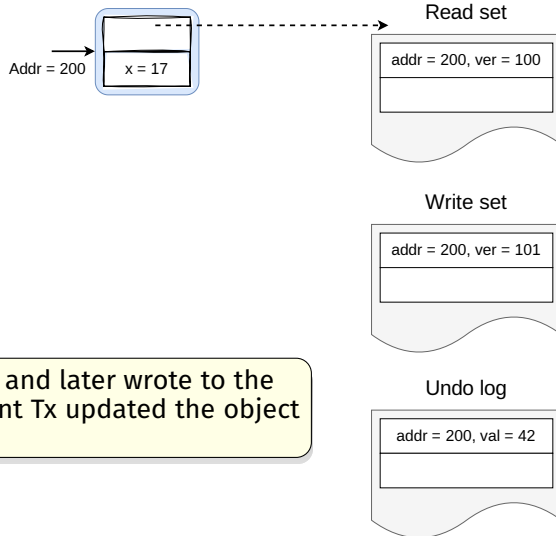


Read set



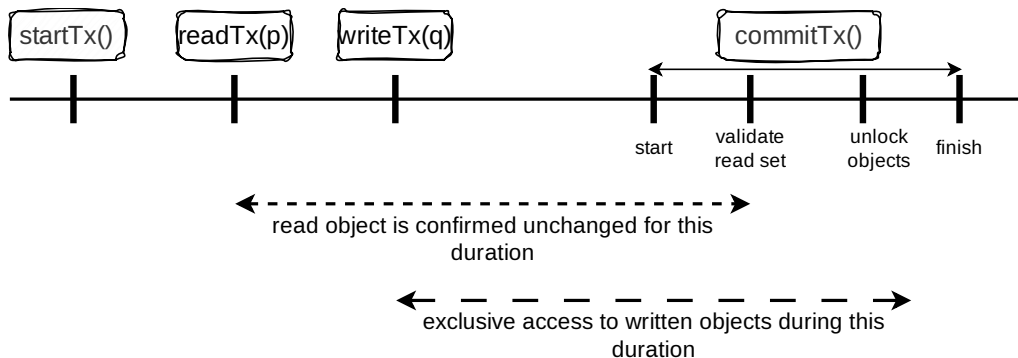
Transaction is owned by some other Tx when the current reader Tx tries to commit

Conflict Between Read and Write from Addr=200



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

Providing Atomicity using Versioned Locks



⚠ Version overflow

- Theoretical concern, is a practical concern if the metadata is “packed”
 - (i) Globally renumber objects if overflow is rare
 - (ii) 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

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

Explicitly Transactional

- `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

Implicitly Transactional

- `begin_transaction`
- `end_transaction`

All memory accesses are transactional

Possible ISA Extensions

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

Explicitly Transactional

- `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

Implicitly Transactional

- `begin_transaction`
- `end_transaction`

Which one is advantageous?

Explicitly vs Implicitly Transactional HTMs

Explicitly Transactional

- Provides flexibility to log relevant transactional memory locations
 - ▶ Reduces read and write set sizes
- May require multiple library versions—transactional and non-transactional
 - ▶ Limits reuse of legacy libraries in HTMs

Implicitly Transactional

- Larger read and write sets
- Easy to reuse software libraries

💡 Tracking read and write sets

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

i Conflict detection

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

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)

- Aims to improve the performance of lock-based critical sections while maintaining the lock-based programming model
- TSX hardware can dynamically determine whether threads need to serialize lock-protected critical sections
 - ▶ Serialize only when required, elide locks when unnecessary
- Supported by Intel in selected series based on Haswell microarchitecture

TSX operation

- (i) Optimistically executes critical sections eliding lock operations
- (ii) Commit if the Tx executes successfully
- (iii) Otherwise abort — discard all updates, restore architectural state, and resume execution
- (iv) Resumed execution may retry with lock elision or fall back to locking

Hardware Lock Elision (HLE)

- xacquire
- xrelease
- Compatible with the lock-based programming model
- Extends HTM support to legacy hardware
- Processors without HLE support ignore these prefixes

Restricted Transactional Memory (RTM)

- xbegin
- xend
- xabort
- New ISA extension
- Processors without RTM support will generate a #UD exception
- Programmer must provide a non-transactional path for aborts

Other Instructions

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

XABORT Allows explicitly aborting a transaction

XSUSLDTRK Suspends tracking loads, allows excluding addresses from the read set

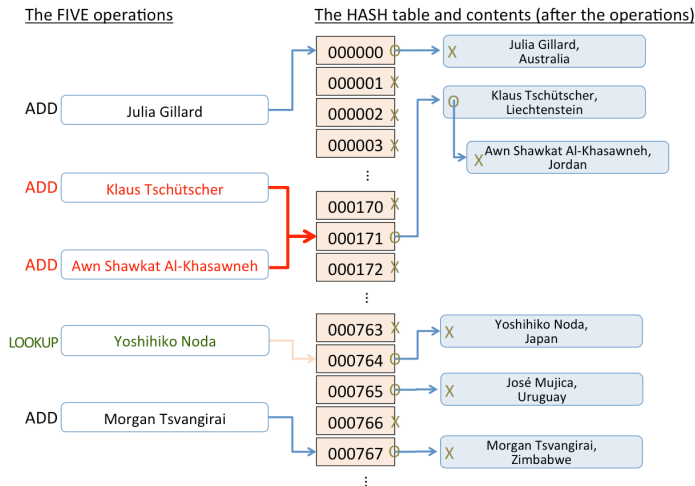
XRESLDTRK Resume tracking loads

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

HLE enabled software has the same forward progress guarantees as the underlying non-HLE lock-based execution

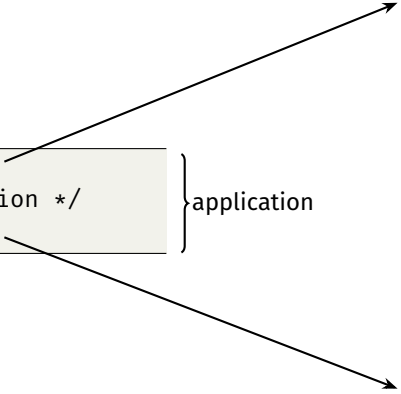
Goal with Intel TSX



Lock Acquire Code

```
1 acquire(mutex);  
2 /* critical section */  
3 release(mutex);
```

} application



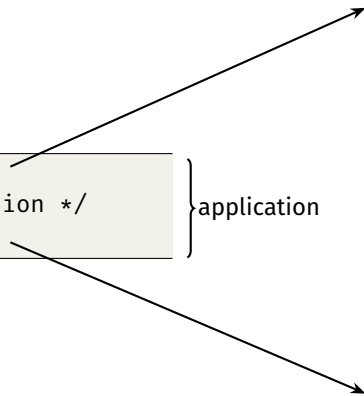
```
1 mov eax, 1  
2 Try: lock xchg mutex, eax  
3 cmp eax, 0  
4 jz Success  
5 Spin: pause  
6 cmp mutex, 1  
7 jz Spin  
8 jmp Try
```

```
1 mov mutex, 0
```

HLE Interface

```
1 acquire(mutex);  
2 /* critical section */  
3 release(mutex);
```

} application



```
1 mov eax, 1  
2 Try: xacquire lock xchg mutex, eax  
3 cmp eax, 0  
4 jz Success  
5 Spin: pause  
6 cmp mutex, 1  
7 jz Spin  
8 jmp Try
```

```
1 xrelease mov mutex, 0
```

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

RTM Interface

```
1 Retry: xbegin Abort
2       cmp mutex, 0
3       jz Success
4       xabort $0xff
```

```
5
6 Abort:
```

```
7     // check eax and do retry
8     // policy actually acquire
9     // lock or wait to retry
10    ...
```

```
1     cmp mutex, 0
2     jnz Rel
3     xend
```

acquire(mutex)

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

```
5
6 Spin: pause
7       cmp mutex, 1
8       jz Spin
9       jmp Try
```

```
1 Rel:  mov mutex, 0
2
3
```

release(mutex)

Aborts in TSX

- Conflicting accesses from different cores (data, locks, false sharing)
 - ▶ TSX maintains read and 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

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

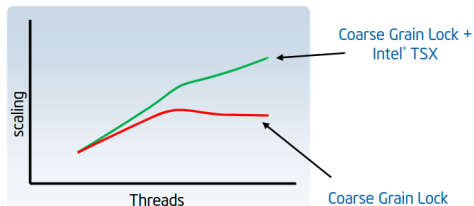
- Every detail is not known
 - ▶ Read and write sets are at cache line granularity
 - ▶ Uses L1 data cache as the storage
- Conflict detection is through cache coherence protocol

TSX caveats

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

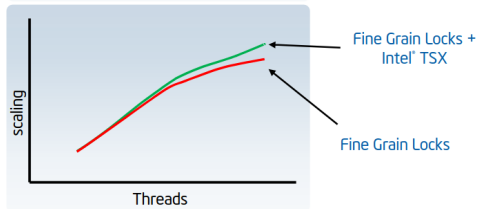
Applying Intel TSX

Application with Coarse Grain Lock



Application re-written with Finer Grain Locks

An example of secondary benefits of Intel TSX



Adoption of TSX-like Concepts

- 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
 - ▶ Requires compiling Glibc with the `-enable-lock-elision=yes` parameter
- 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. Chapters 1, 2.1–2.2, 4.1–4.2, 5.1, 2nd edition, Springer Cham.



Web Resources About Intel® Transactional Synchronization Extensions



Intel®64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, Chapter 17.



Intel®64 and IA-32 Architectures Optimization Reference Manual: Volume 1, Chapter 16.



R. Rajwar and M. Dixon. Intel®Transactional Synchronization Extensions. Intel Developer Forum, 2012.



A. Kleen. Adding lock elision to Linux. Linux Plumbers Conference, 2012.



Dr. Roman Dementiev. Making the Most of Intel®Transactional Synchronisation Extension.



R. Rajwar. Going Under the Hood with Intel's Next Generation Microarchitecture Codename Haswell. QCon 2012.