

# Lock-free programming and transactional memory

Amitabha Roy

2/4

# Overview

Multi-core systems, and concurrent programming without locks

Transactional memory for managing shared-memory data structures

Integrating transactional memory into a modern, object-oriented programming language

Making sense of transactional memory: combining transactions with libraries, locking, and IO

# Example STM primitive API

- `tx = TxStart()`
- `b = TxCommit(tx)`
- `b = TxValidate(tx)`
- `TxAbort(tx)`
  
- `v = TxRead(tx, addr)`
- `TxWrite(tx, addr, v)`

Transaction management: start a transaction, attempt to commit one, force a transaction to abort, test if a transaction is valid

All transacted memory accesses use explicit TxRead / TxWrite operations

# Example: a double-ended queue

```
class Q {
    QElem leftSentinel;
    QElem rightSentinel;

    void pushLeft(int item) {
        QElem e = new QElem(item);
        do {
            TxStart();
            TxWrite(&e.right, TxRead(&this.leftSentinel.right));
            TxWrite(&e.left, this.leftSentinel);
            TxWrite(&TxRead(&this.leftSentinel.right).left, e);
            TxWrite(&this.leftSentinel.right, e);
        } while (!TxCommit());
    }

    ...
}
```

# Desirable properties for STM

- Fast:
  - Individual primitives should be fast
- Scalable:
  - Primitives should scale internally (e.g. operations on different locations don't conflict in memory)
  - Transactions build over these primitives should scale (e.g. one transaction should not be forced to roll-back by non-conflicting accesses in another transaction)
- Predictable:
  - Cases where the implementation introduces costs or false conflicts should be able to be anticipated

# Taxonomy: consistency during tx

- Gold standard:
  - During execution a transaction runs against a consistent view of memory
  - Won't be “tricked” into looping, etc.
  - “Opacity”
- What are the advantages / disadvantages when compared with an implementation giving weaker guarantees?

# Taxonomy: lazy/eager versioning

- We need some way to manage the tentative updates that a transaction is making
  - Where are they stored?
  - How does the implementation find them (so a transaction's read sees an earlier write)?
- Lazy versioning: only make “real” updates when a transaction commits
- Eager versioning: make updates as a transaction runs, roll them back on abort
- What are the advantages, disadvantages?

# Taxonomy: lazy/eager conflict detection

- We need to detect when two transactions conflict with one another
- Lazy conflict detection: detect conflicts at commit time
- Eager conflict detection: detect conflicts as transactions run
- Again, what are the advantages, disadvantages?

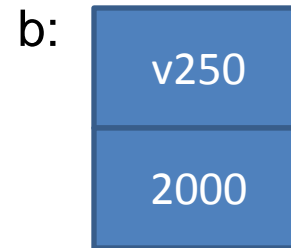
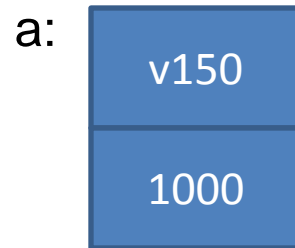
# Taxonomy: word/object based

- What granularity are conflicts detected at?
- Object-based:
  - Access to programmer-defined structures (e.g. objects)
- Word-based:
  - Access to words (or sets of words, e.g. cache lines)
  - Possibly after mapping under a hash function
- What are the advantages and disadvantages of these approaches?

# Bartok-STM

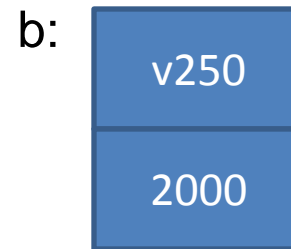
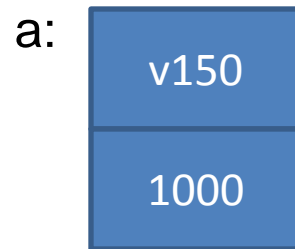
- Use per-object meta-data (“TMWs”)
- Each TMW is either:
  - Locked, holding a pointer to the transaction that has the object open for update
  - Available, holding a version number indicating how many times the object has been locked
- Writers eagerly lock TMWs to gain access to the object, using eager version management
  - Maintain an undo log in case of roll-back
- Readers log the version numbers they see and perform lazy conflict detection at commit time

# Example: uncontended swap



```
void Swap(int *a, int *b)
{
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```

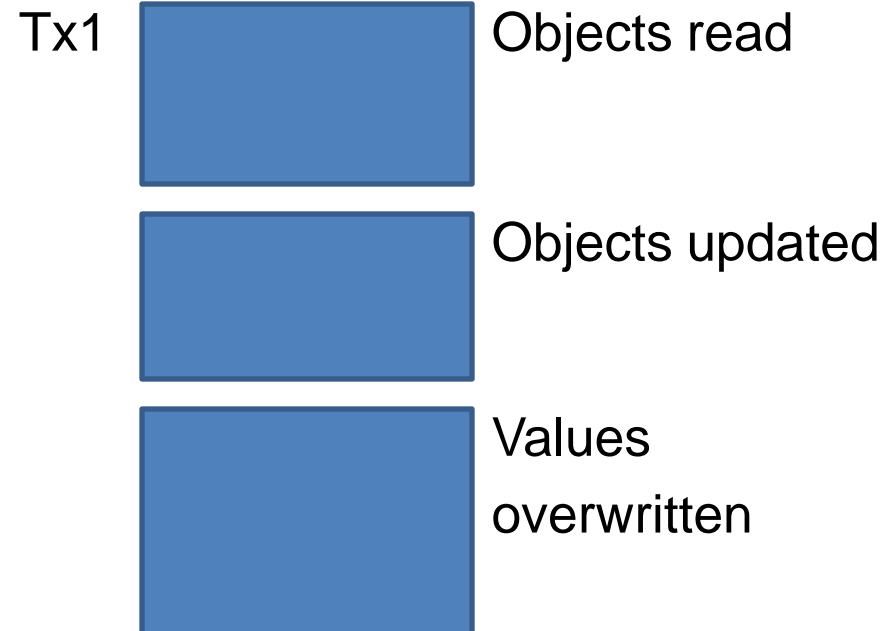
# Example: uncontended swap



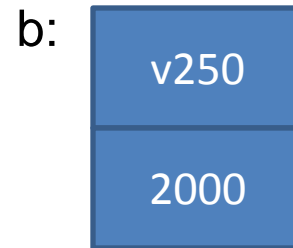
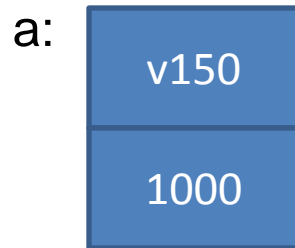
```

void Swap(int *a, int *b)
{
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}

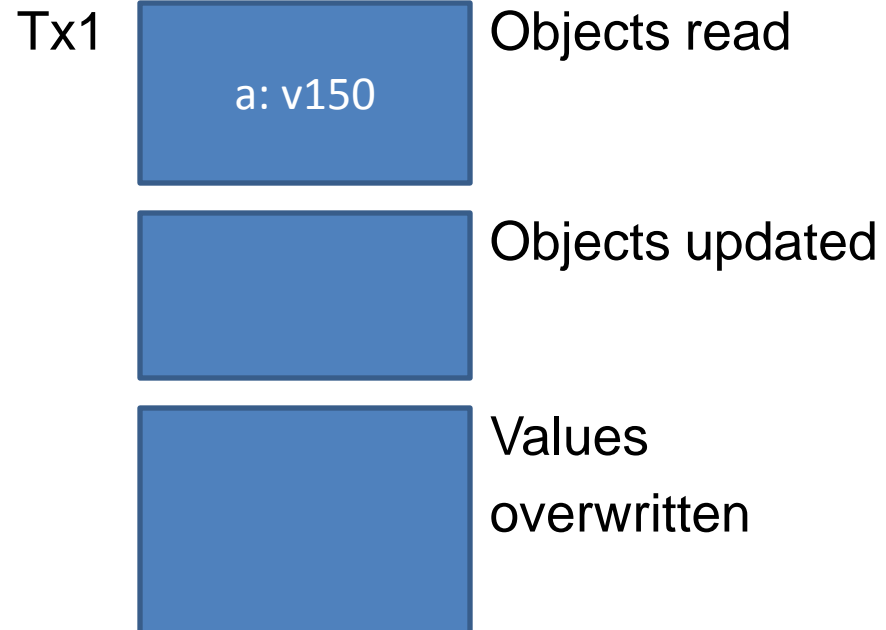
```



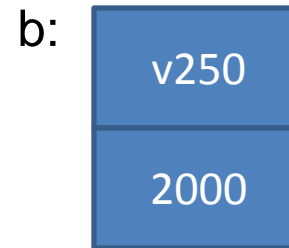
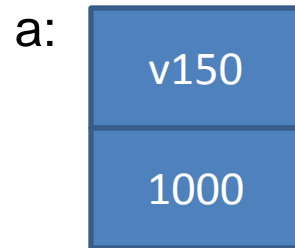
# Example: uncontended swap



```
void Swap(int *a, int *b)
{
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```

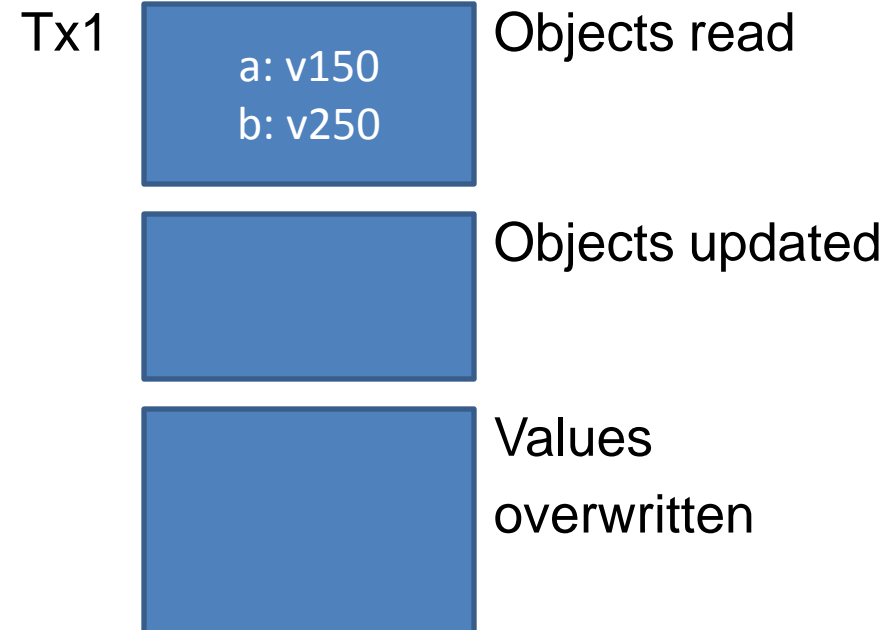


# Example: uncontended swap

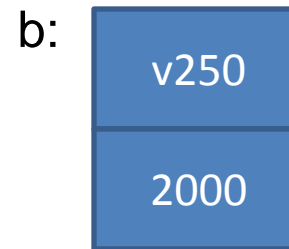
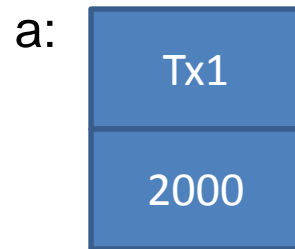


```

void Swap(int *a, int *b)
{
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
    
```



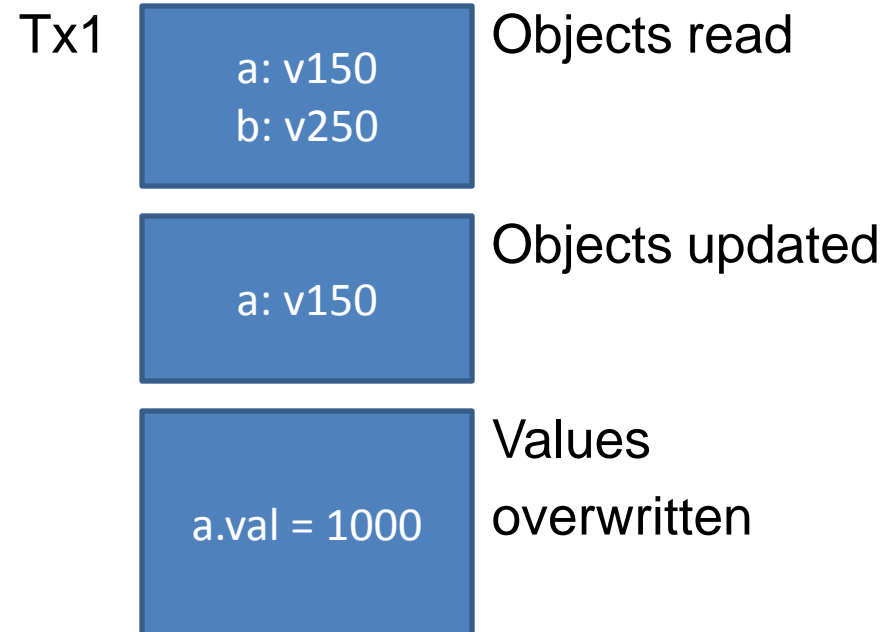
# Example: uncontended swap



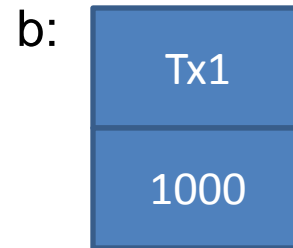
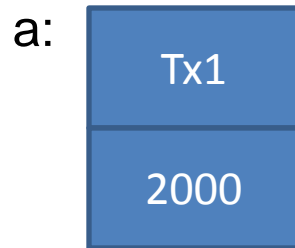
```

void Swap(int *a, int *b)
{
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}

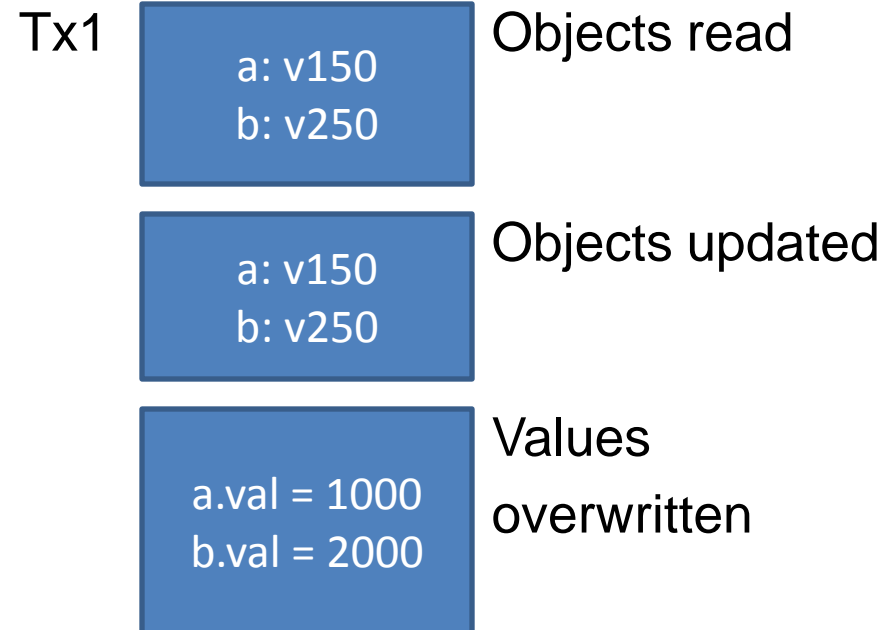
```



# Example: uncontended swap

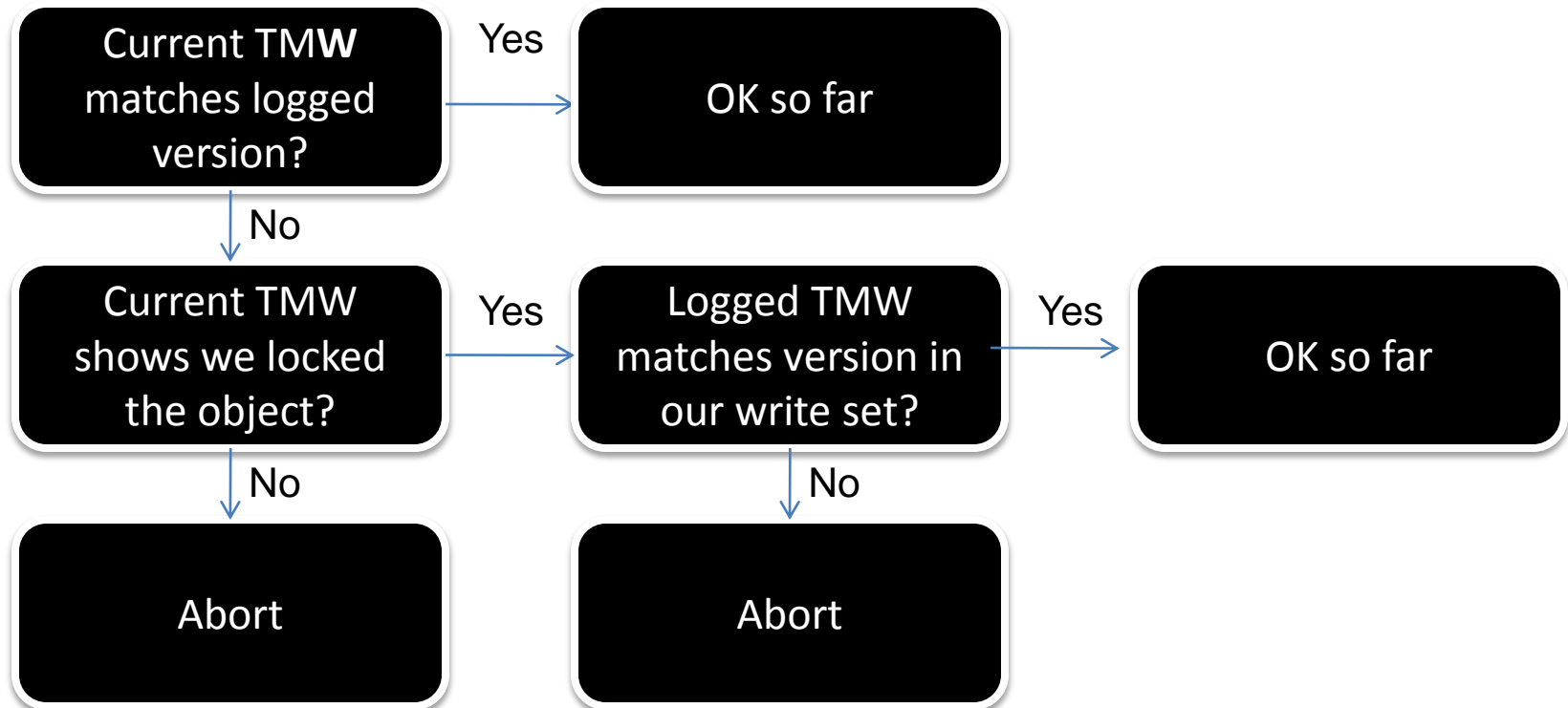


```
void Swap(int *a, int *b)
{
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```

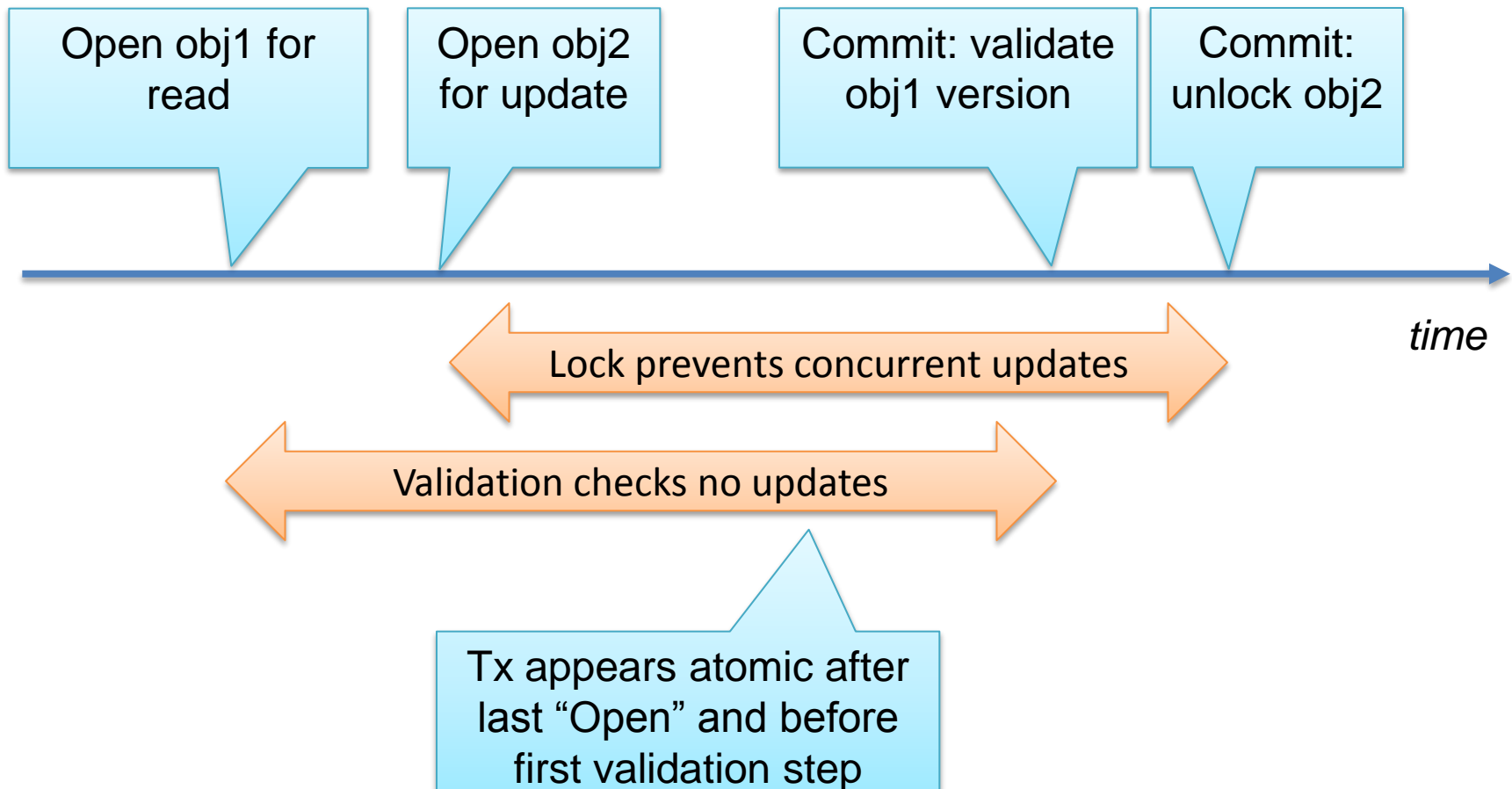


# Commit in Bartok-STM

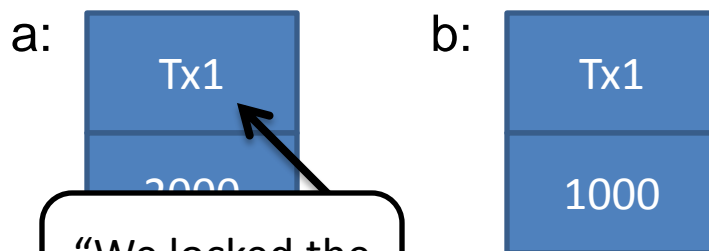
Iterate over  
the read set:



# Correctness sketch

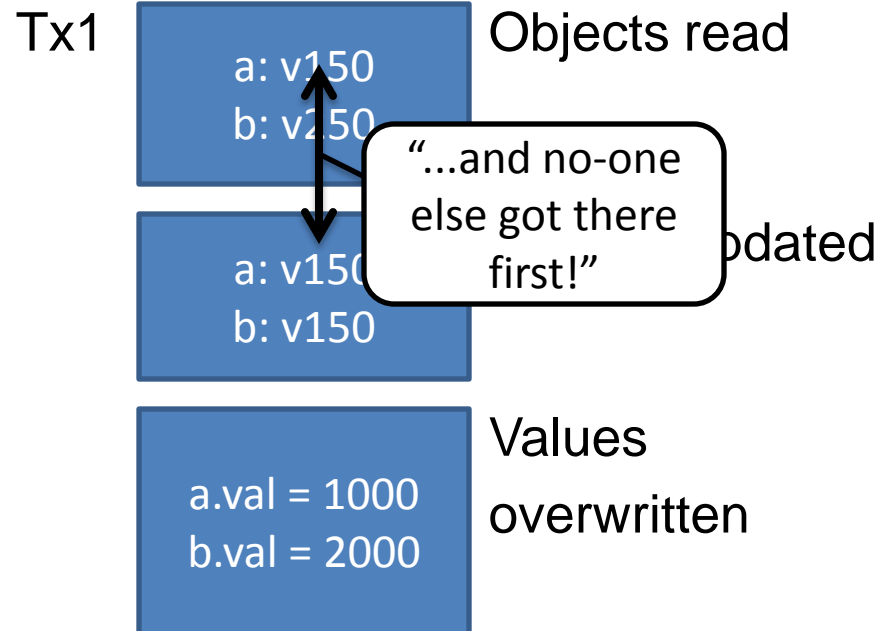


# Example: uncontended swap



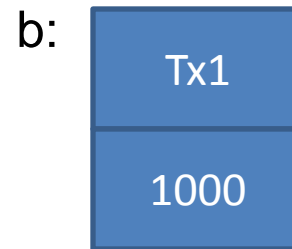
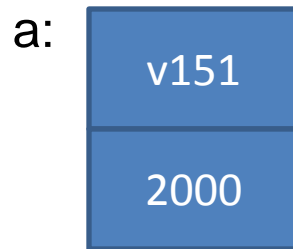
"We locked the object..."

```
void Swap(int *a, int *b)
{
    do {
        tx = TxStart();
        va = TxRead(tx, &a);
        vb = TxRead(tx, &b);
        TxWrite(tx, &a, vb);
        TxWrite(tx, &b, va);
    } while (!TxCommit());
}
```



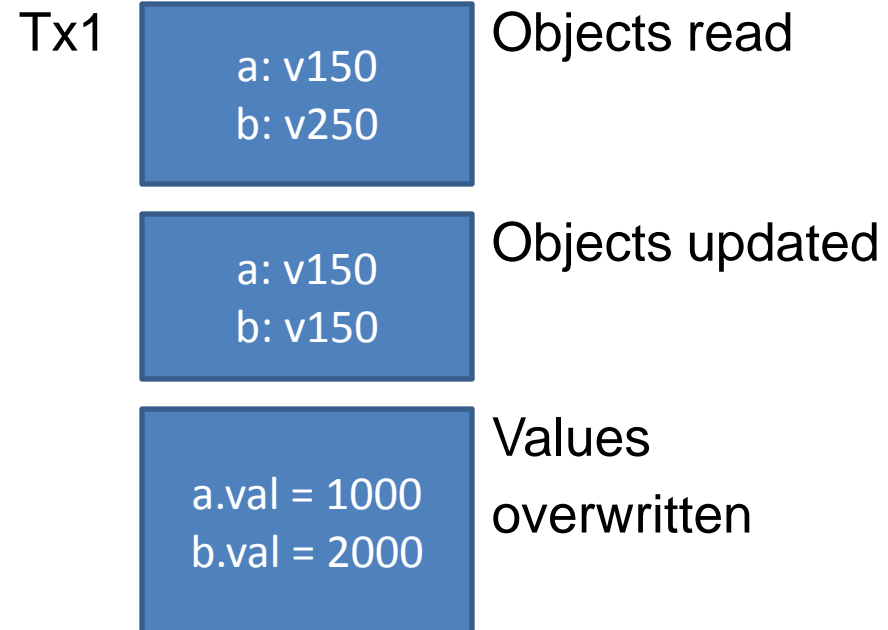
"...and no-one else got there first!"

# Example: uncontended swap

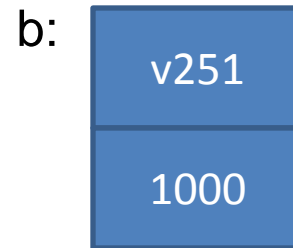
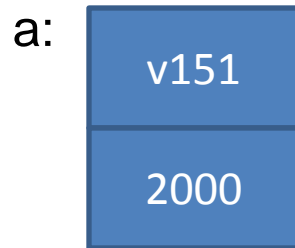


```


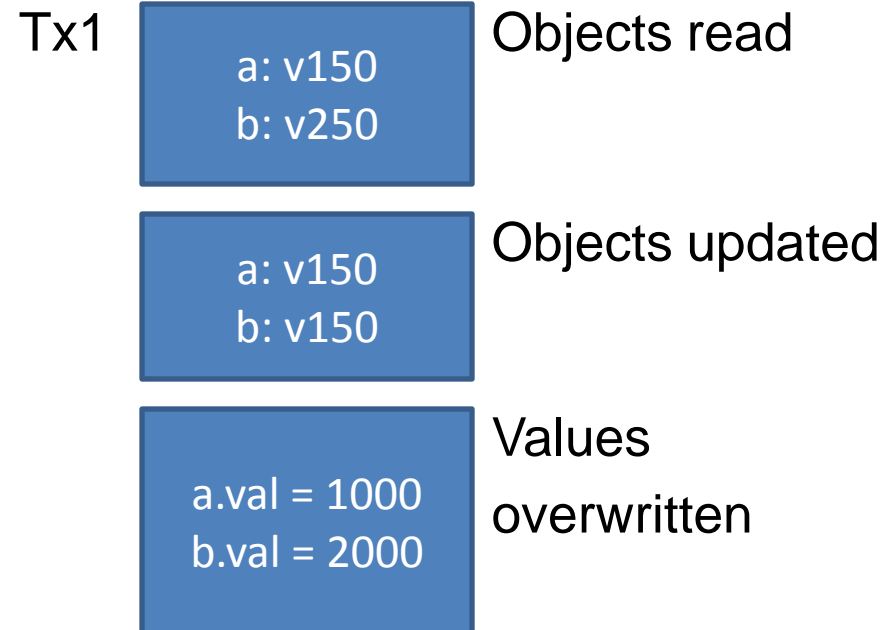
void Swap(int *a, int *b)
{
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
    
```



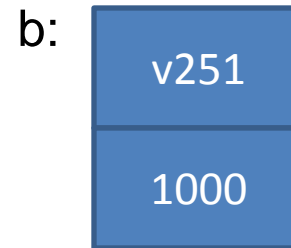
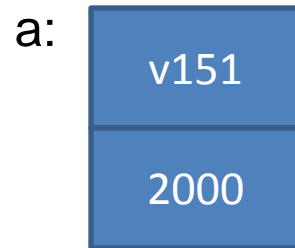
# Example: uncontended swap




```
void Swap(int *a, int *b)
{
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
}
```

# Example: uncontended swap



```
void Swap(int *a, int *b)
{
  do {
    tx = TxStart();
    va = TxRead(tx, &a);
    vb = TxRead(tx, &b);
    TxWrite(tx, &a, vb);
    TxWrite(tx, &b, va);
  } while (!TxCommit());
```



# Tx-tx interaction in Bartok-STM

- Read-read: no problem, both readers see the same version number and verify it at commit time
- Read-write: reader sees that the writer has the object locked. Reader always defers to writer
- Write-write: competition for lock serializes writers (drop locks, then spin to avoid deadlock)

# Bartok-STM

- Designed to work well on low-contention workloads
  - Eager version management to reduce commit costs
  - Eager locking to support eager version management
- Primitives do not guarantee that transactions see a consistent view of the heap while running
  - Can be sandboxed in managed code...
  - ...harder in native code

# TL2

- Global “clock” incremented upon every commit
- Tx log the global clock when they start – “read version”,  $rv$
- Tx perform lazy version management
- On reads, check the object’s timestamp  $\leq rv$
- At commit:
  1. lock tentatively-updated locations

# Example: uncontended swap

500

a: 

v450
1000

b: 

v350
2000

```
void Swap(int *a, int *b)
{
do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
} while (!CommitTx());
}
```

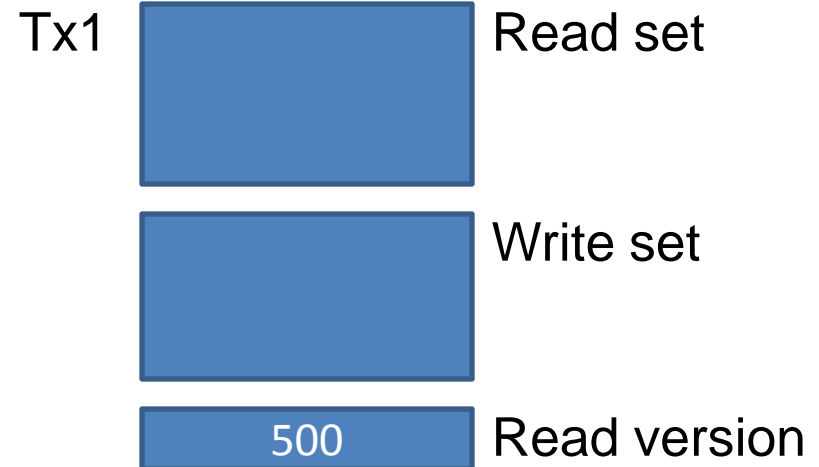
# Example: uncontended swap

500

a:  
v450  
1000

b:  
v350  
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



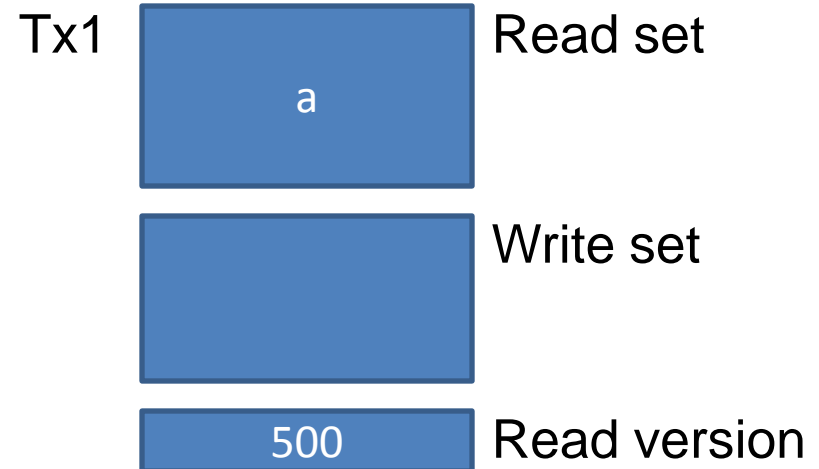
# Example: uncontended swap

500

a:  
v450  
1000

b:  
v350  
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



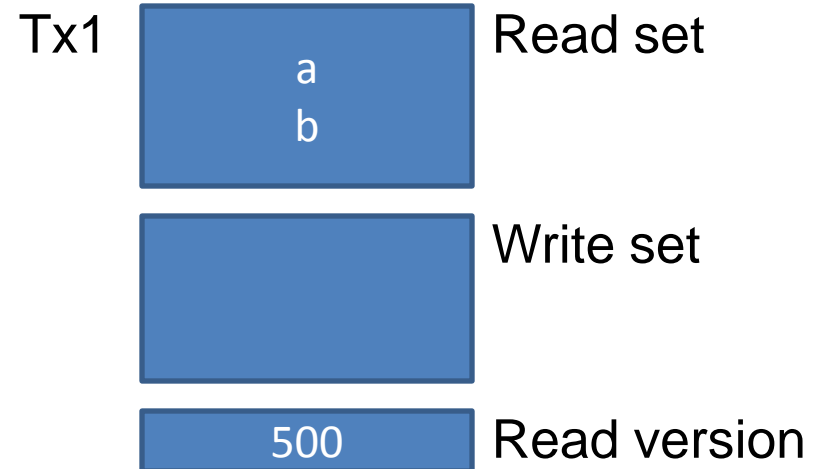
# Example: uncontended swap

500

a:  
v450  
1000

b:  
v350  
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



# Example: uncontended swap

500

a: 

v450
1000

b: 

v350
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000	Write set
500	Read version

# Example: uncontended swap

500

a:  
v450  
1000

b:  
v350  
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version

# Example: uncontended swap

600

a: Tx1,  
v450  
1000

b: v350  
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version

# Example: uncontended swap

600

a: Tx1,  
v450  
1000

b: Tx1,  
v350  
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version

# Example: uncontended swap

601

a: Tx1,  
v450  
-----  
1000

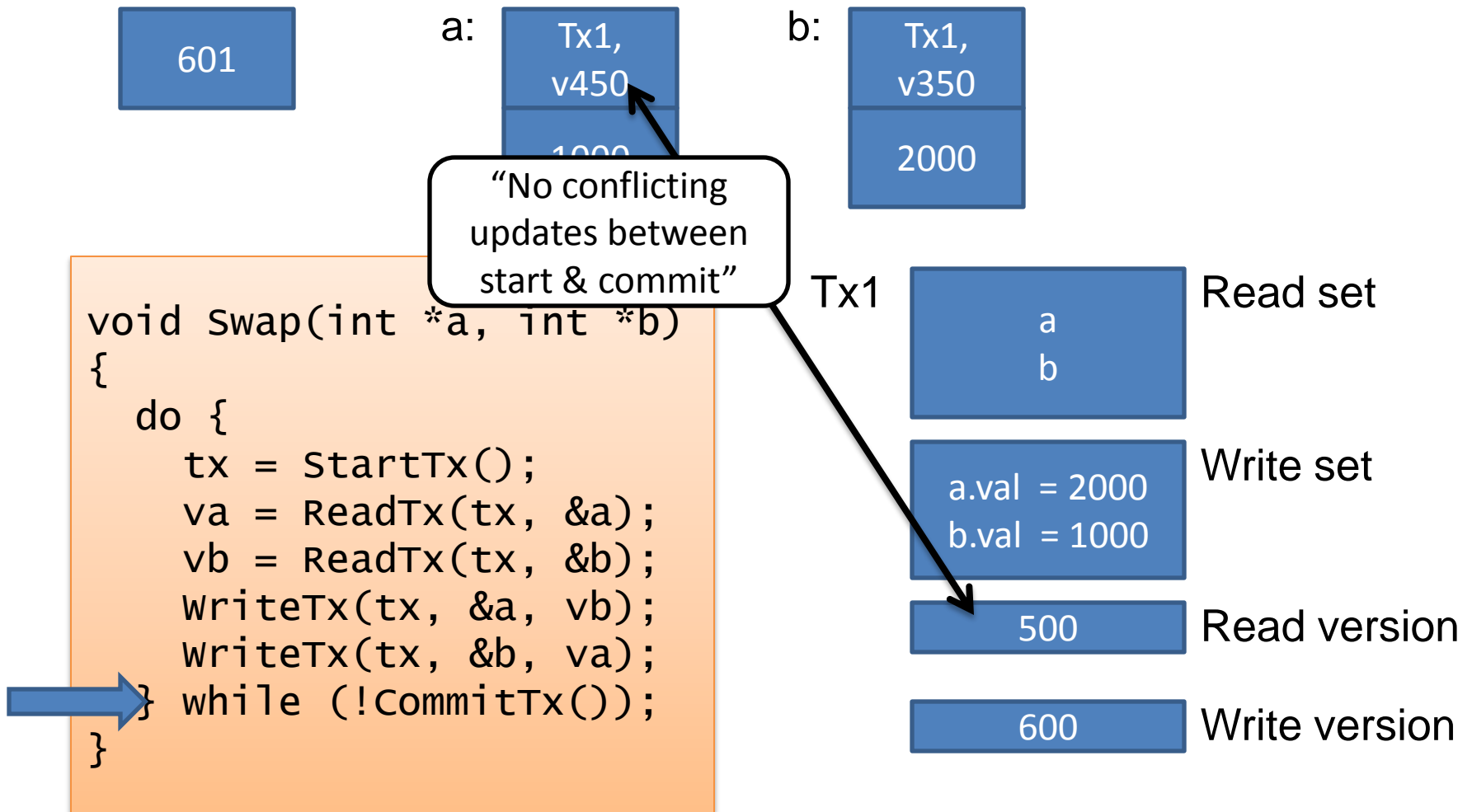
b: Tx1,  
v350  
-----  
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version
600	Write version

# Example: uncontended swap



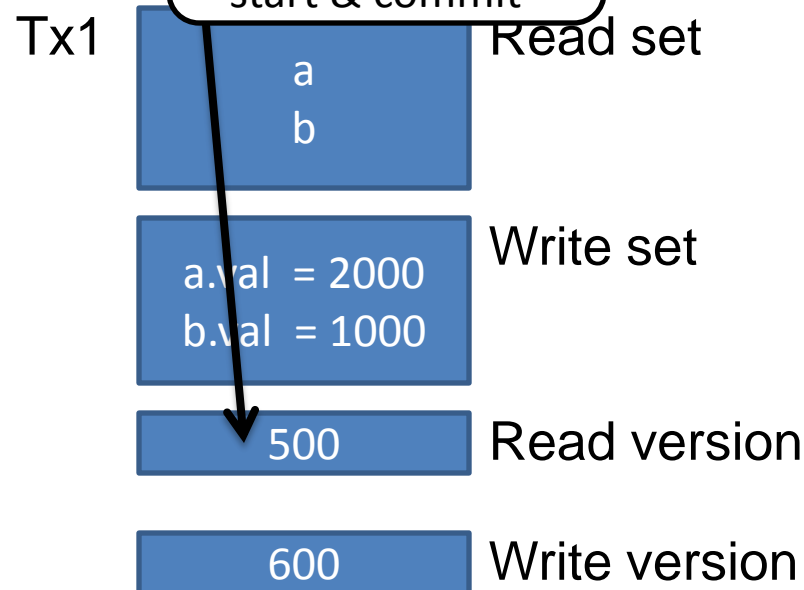
# Example: uncontended swap

601

a: Tx1, v450  
1000

b: Tx1, v350  
2000

"No conflicting updates between start & commit"



```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```



# Example: uncontended swap

601

a: Tx1,  
v450  
2000

b: Tx1,  
v350  
2000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version
600	Write version

# Example: uncontended swap

601

a: Tx1,  
v450  
2000

b: Tx1,  
v350  
1000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version
600	Write version

# Example: uncontended swap

601

a:  
v600  
2000

b:  
v600  
1000

```
void Swap(int *a, int *b)
{
  do {
    tx = StartTx();
    va = ReadTx(tx, &a);
    vb = ReadTx(tx, &b);
    writeTx(tx, &a, vb);
    writeTx(tx, &b, va);
  } while (!CommitTx());
}
```

Tx1

a b	Read set
a.val = 2000 b.val = 1000	Write set
500	Read version
600	Write version

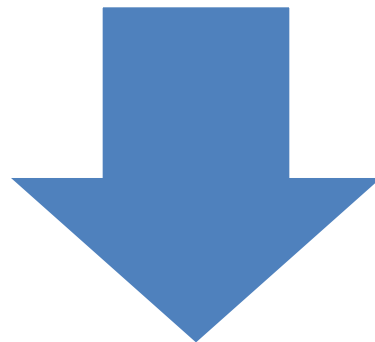
# Tx-tx interaction in TL2

- Read-read: no problem, both readers see the same version number and verify it at commit time
- Reader-tentative-write: commit-time locking means that the reader can commit if they finish before the writer
- Reader-committed-writer: reader aborts
- Write-write: competition for lock serializes writers at commit time

# Low-level optimizations in TL2

- Accelerate look-aside into log:
  - Use Bloom filter to check if the location may be in the write set
  - Can use hashing mechanisms to reduce log searching
- Don't need the read set in read-only transactions
- Don't need to validate if  $\text{write\_version} = \text{read\_version} + 1$
- Reduce contention on the global version number:
  - Combine all version numbers with a thread ID of the last modifier
  - Only increment global version number if it differs from our thread's last write-version
  - Otherwise the global version number and our thread ID are globally unique

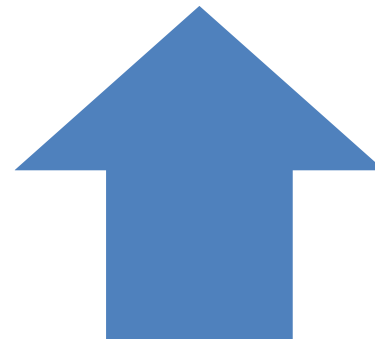
# Performance trade-offs & dangers



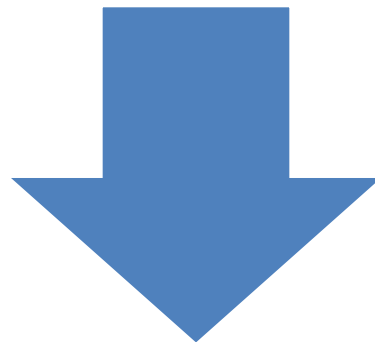
Improve concurrency between transactions (e.g. detect that an execution is linearizable, even though conflicting transactions overlapped in time)



Improve the speed of the STM primitives, e.g. by performing less book-keeping



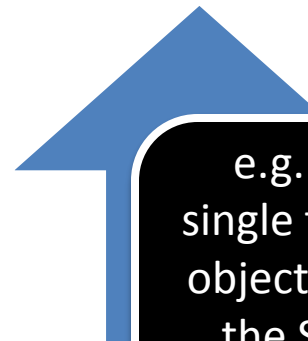
# Performance trade-offs & dangers



Improve concurrency between transactions (e.g. detect that an execution is linearizable, even though conflicting transactions overlapped in time)



Improve the speed of the STM primitives, e.g. by performing less book-keeping



e.g. Bartok-STM allows a single tentative writer to each object: is the simplification in the STM worth the loss of concurrency between transactions?

# Summary

- Common design features:
  - Use locking to arbitrate between writers (e.g. compare with the complexity of early non-blocking commits) and version numbers to detect conflicts on reads
  - Correctness arguments can both be based on identifying a point at which a transaction appears atomic
- TL2's use of a global version number gives stronger guarantees to the programmer building over it
- Many implementation variants possible, e.g. addr-to-TMW via a hash function, writers linked from an object header, indirection to object versions