Lecture 20:

Transactional Memory

Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013

Slide credit

- Many of the slides in today's talk are borrowed from Professor Christos Kozyrakis (Stanford University)

Raising level of abstraction for synchronization

- Machine-level atomic operations:
  - Fetch-and-op, test-and-set, compare-and-swap

- We used these atomic operations to construct higher level synchronization primitives:
  - Locks, barriers
  - We've seen how it can be challenging to produce correct programs using these primitives (violate atomicity, deadlock, etc.)

- Today: raising level of abstraction for synchronization further
  - Transactional memory

What you should know

- What a transaction is
- The difference between an `atomic` block and locks
- The basic design space of transactional memory implementations
  - Data versioning policy
  - Conflict detection policy
  - Granularity of detection
- Understand the basics of HW implementations of transactional memory (consider how it relates to coherence protocol implementations we've discussed previously in the course)

Review: ensuring atomicity via locks

```
void deposit(Acct account, int amount)
{
   lock(account.lock);
   int tmp = bank.get(account);
   tmp += amount;
   bank.put(account, tmp);
   unlock(account.lock);
}
```

- `Deposit` is a read-modify-write operation: want "deposit" to be atomic with respect to other bank operations on this account.
- Lock/unlock pair is one mechanism to synchronize threads to ensure atomicity (ensure mutual exclusion on the account)

Programming with transactions

```
void deposit(Acct account, int amount)
{
   lock(account.lock);
   int tmp = bank.get(account);
   tmp += amount;
   bank.put(account, tmp);
   unlock(account.lock);
}
```

```
void deposit(Acct account, int amount)
{
   atomic {
     int tmp = bank.get(account);
     tmp += amount;
     bank.put(account, tmp);
   }
}
```

- **Atomic construct is declarative**
  - Programmer states what to do (maintain atomicity of this code), not how to do it
  - No explicit creation or management of locks
- **System implements necessary synchronization to ensure atomicity**
  - Typically using optimistic concurrency
  - System slows down only during true contention (R-W or W-W conflicts)

Declarative vs. imperative abstractions

- Declarative: programmer defines <u>what</u> should be done
  - Execute all these independent 1000 tasks
  - Perform this set of operations atomically

- Imperative: programmer states <u>how</u> it should be done
  - Spawn N worker threads. Assign work to threads by removing work from from shared task queue
  - Acquire a lock, perform operations, release the lock

Transactional Memory (TM)

- **Memory transaction**
  - An atomic & isolated sequence of memory accesses
  - Inspired by database transactions
- **Atomicity (all or nothing)**
  - Upon transaction commit, all memory writes take effect at once
  - On transaction abort, none of the writes appear to take effect
- **Isolation**
  - No other processor can observe writes before commit
- **Serializability**
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed though

Motivating transactional memory

Another example: Java 1.4 HashMap

- Map: Key → Value
  - Implemented as a hash table with linked list per bucket

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

Bad: not thread safe

Good: no lock overhead when synchronization not needed

Synchronized HashMap

- Java 1.4 solution: synchronized layer
  - Convert any map to thread-safe variant
  - Uses explicit, coarse-grained locking specified by programmer

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

- Coarse-grained synchronized HashMap
  - Good: thread-safe, easy to program
  - Bad: limits concurrency, poor scalability

Review from last class

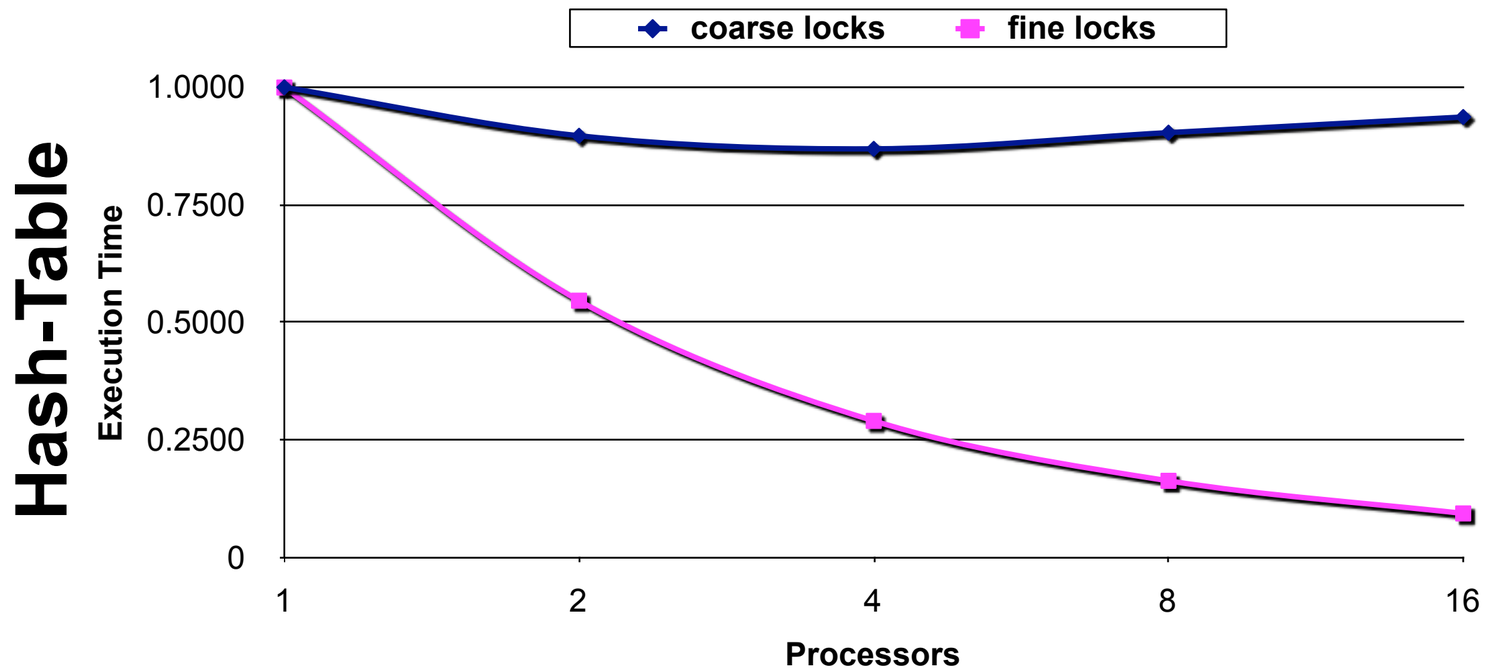
What are better solutions for making hashmap object thread-safe?

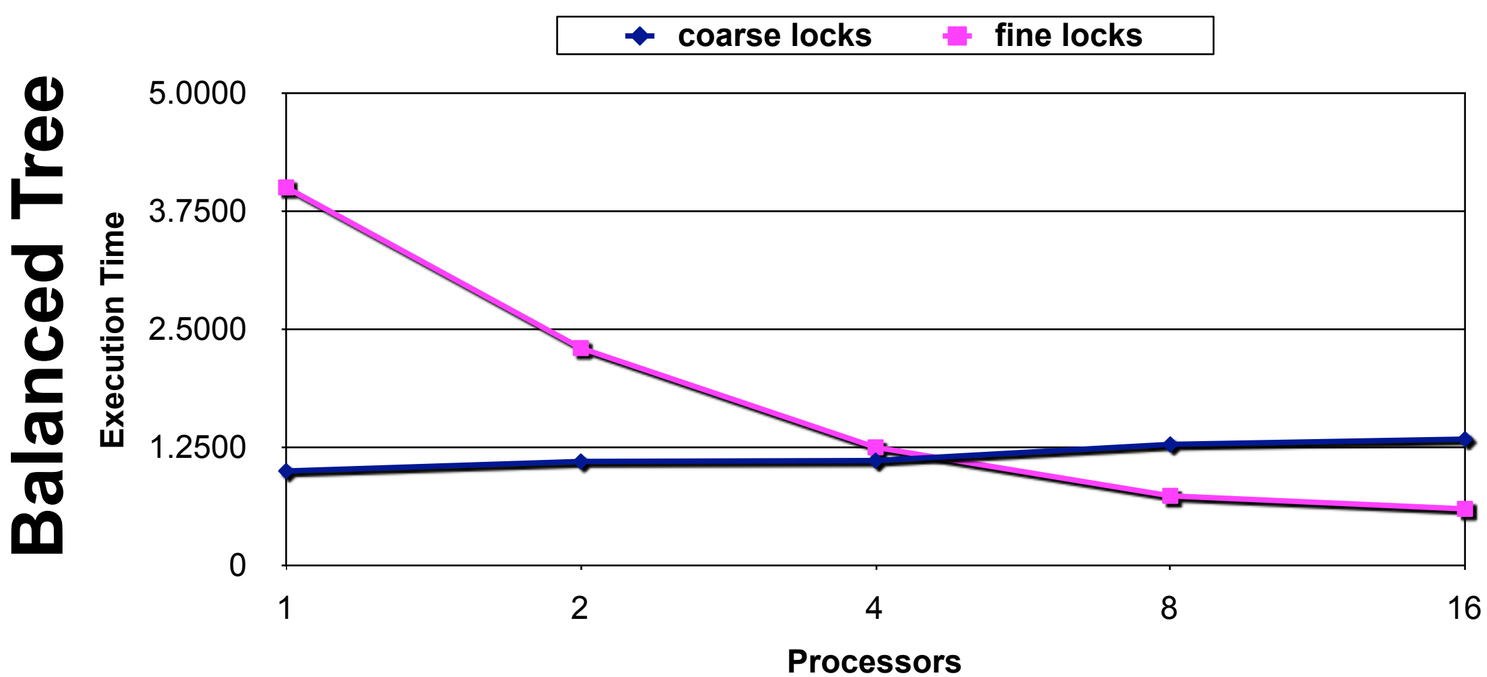
```
public Object get(Object key)  {
    int idx = hash(key);                // compute hash
    HashEntry e = buckets[idx];         // find bucket
    while (e != null)  {                // find element in bucket
        if (equals(key, e.key))
               return e.value;
        e = e.next;
   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

- **Use finer-grained synchronization: e.g., lock per bucket**
  - Now thread safe: but incur lock overhead even if synchronization not needed

Review: performance of fine grained locking

Reduced contention leads to better performance





Transactional HashMap

- Simply enclose all operation in atomic block
  - System ensures atomicity

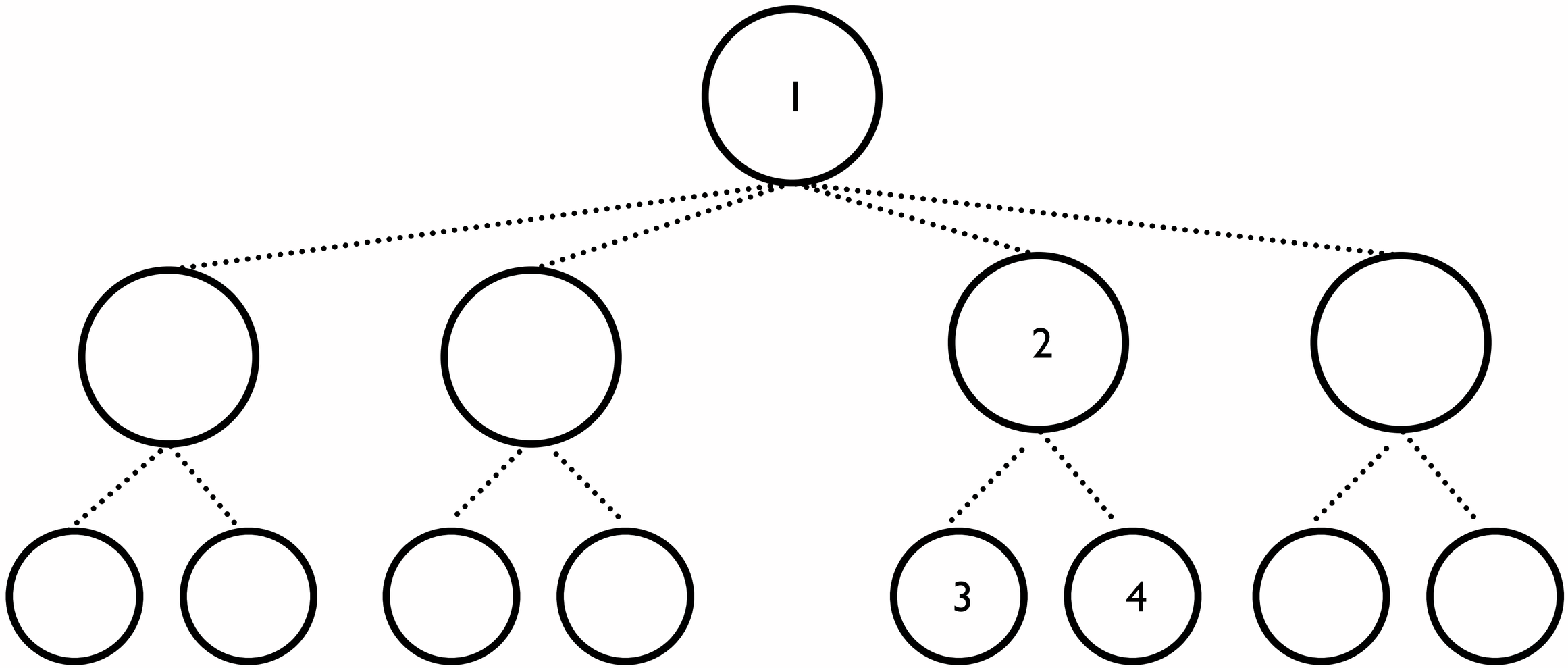
```
public Object get(Object key) {
      atomic {         // System guarantees atomicity
         return m.get(key);
      }
   }
```

- Transactional HashMap

- Good: thread-safe, easy to program
- What about performance & scalability?
  - Depends on the implementation, but typically yes

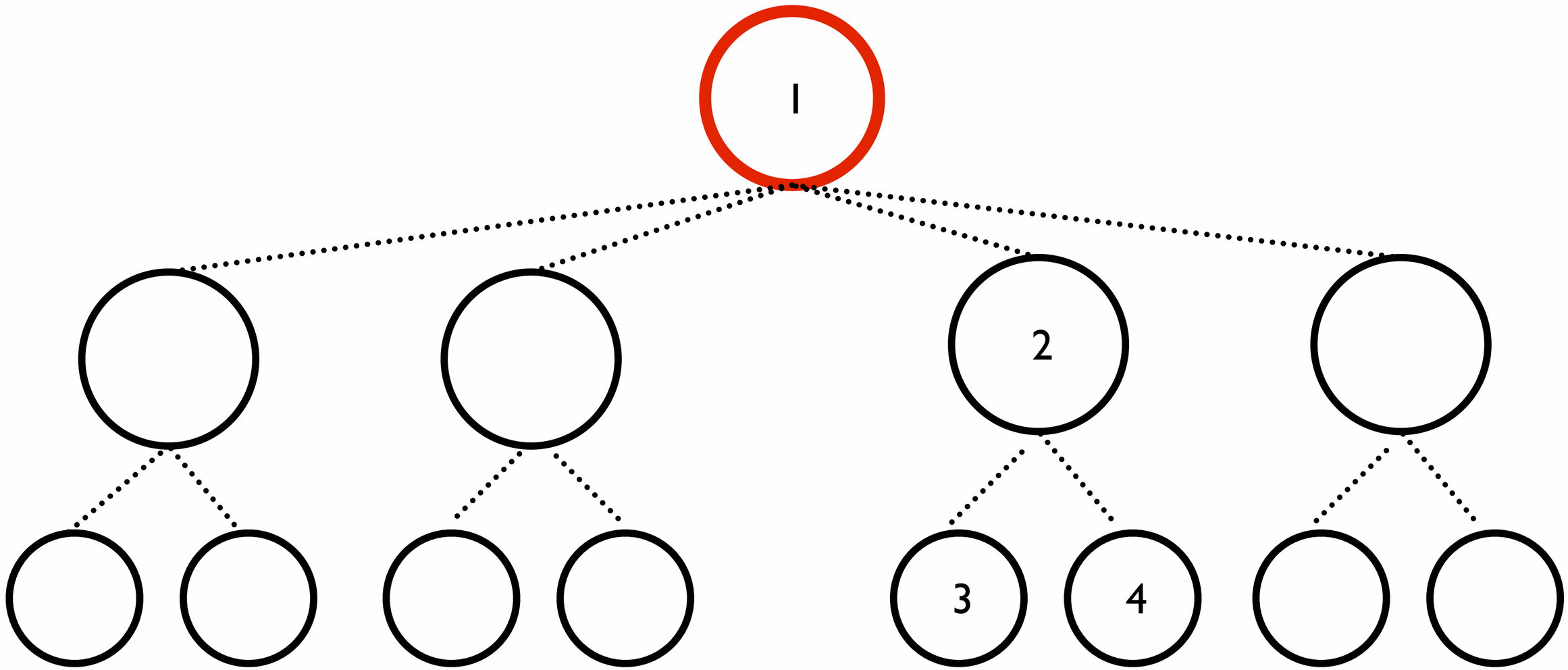
Another example: tree update by two threads

Goal: modify nodes 3 and 4 in a thread-safe way



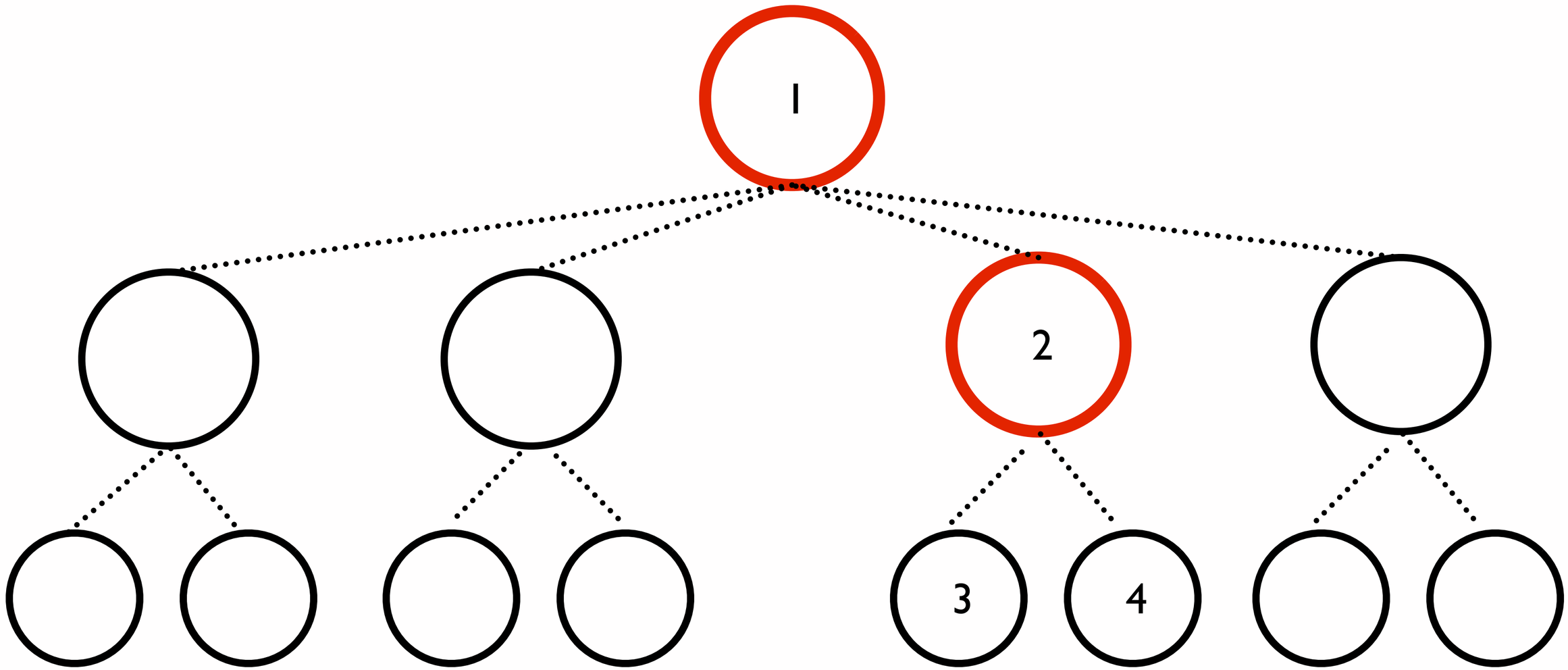
Fine-grained locking example

Hand-over-hand locking



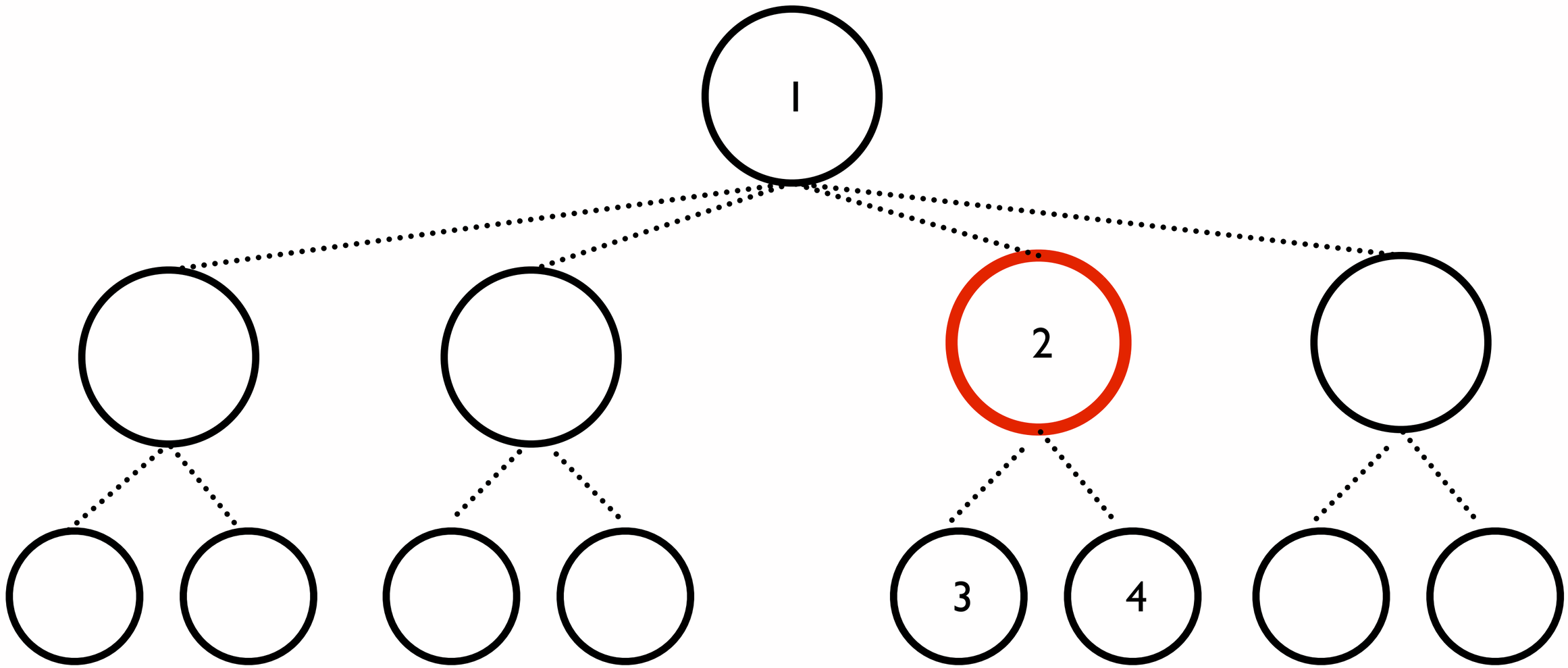
Fine-grained locking example

Hand-over-hand locking



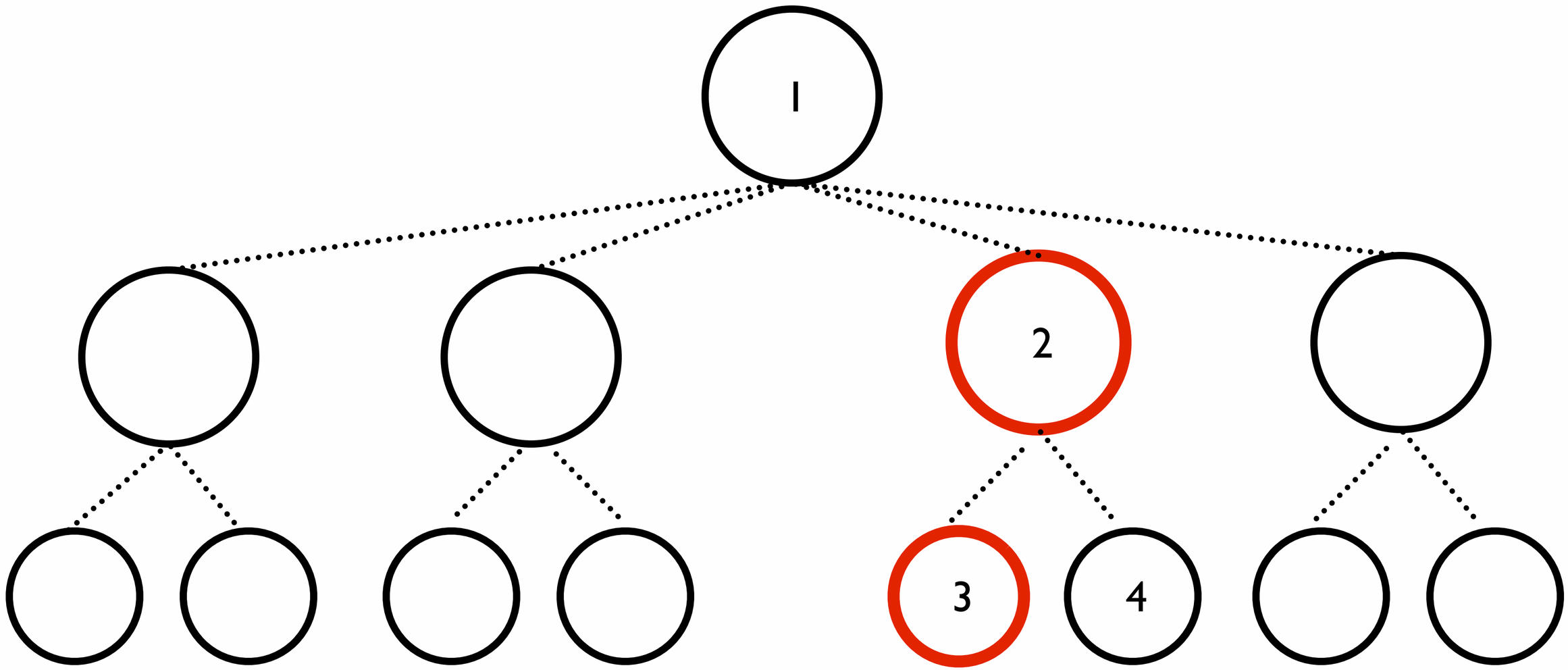
Fine-grained locking example

Hand-over-hand locking



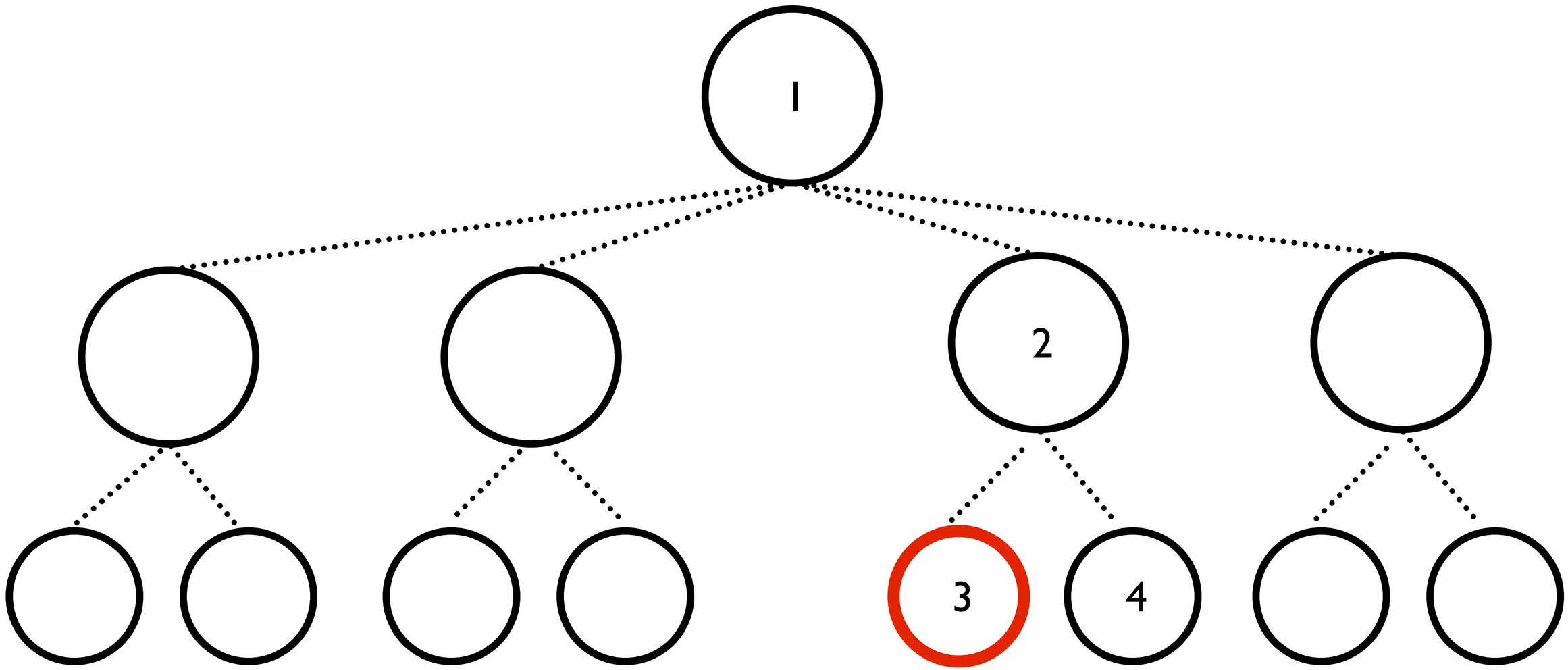
Fine-grained locking example

Hand-over-hand locking



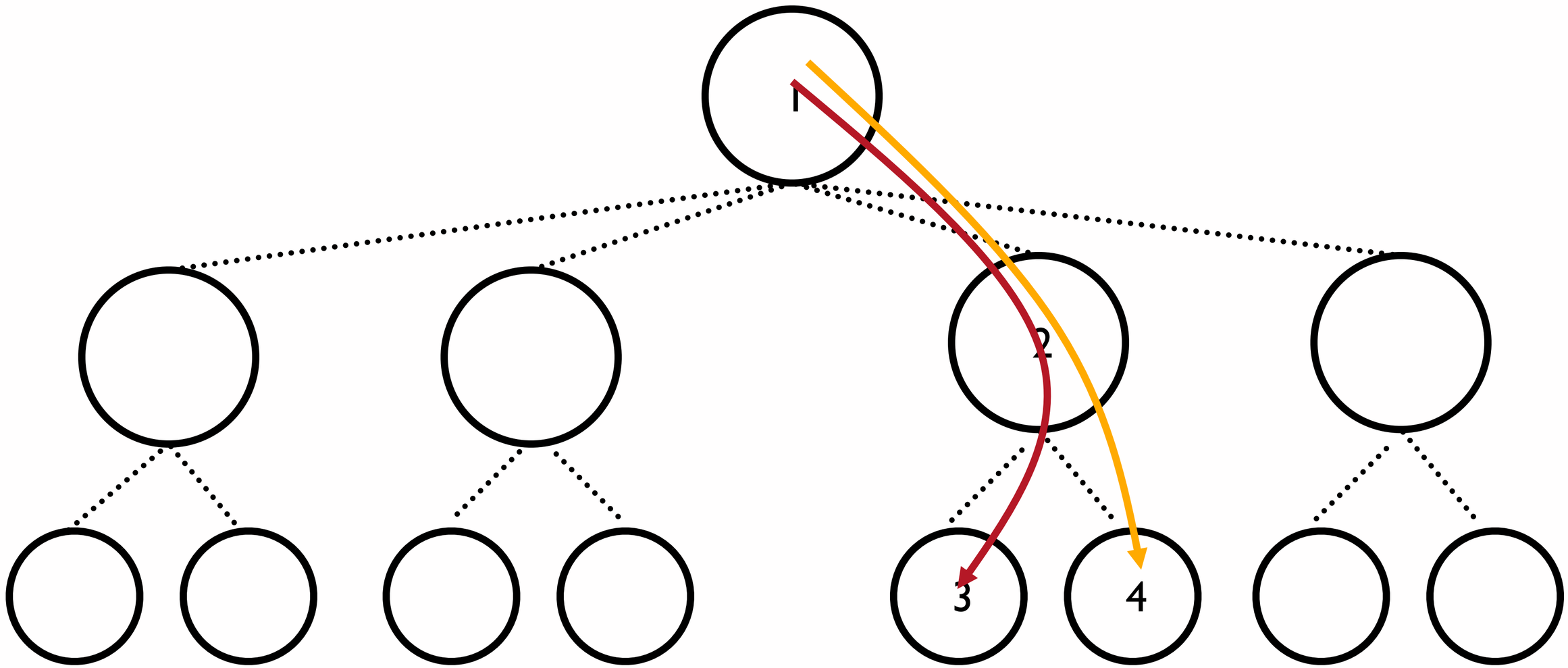
Fine-grained locking example

Hand-over-hand locking



Fine-grained locking example

Hand-over-hand locking

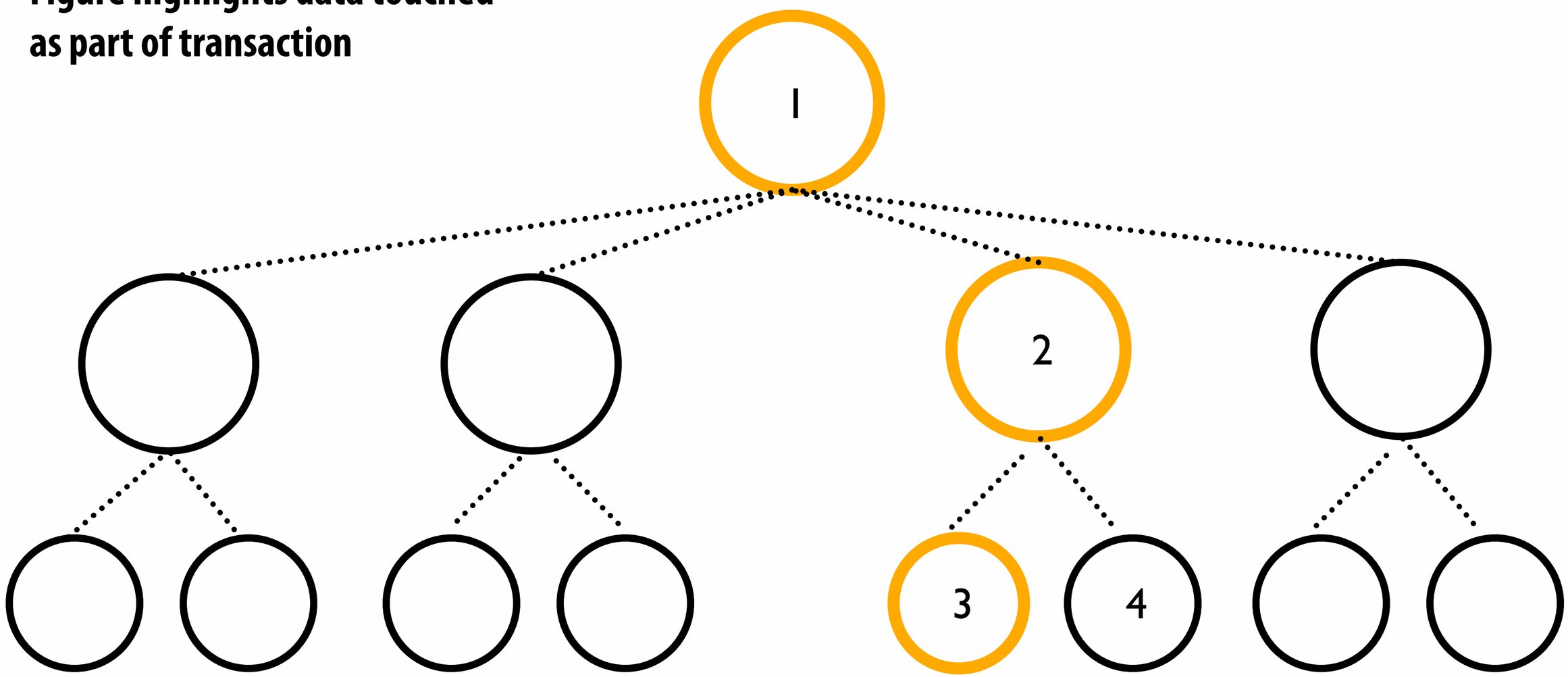


Locking can prevent concurrency

(here: locks on 1 and 2 during update to 3 could delay update to 4)

Transactions example

Figure highlights data touched as part of transaction

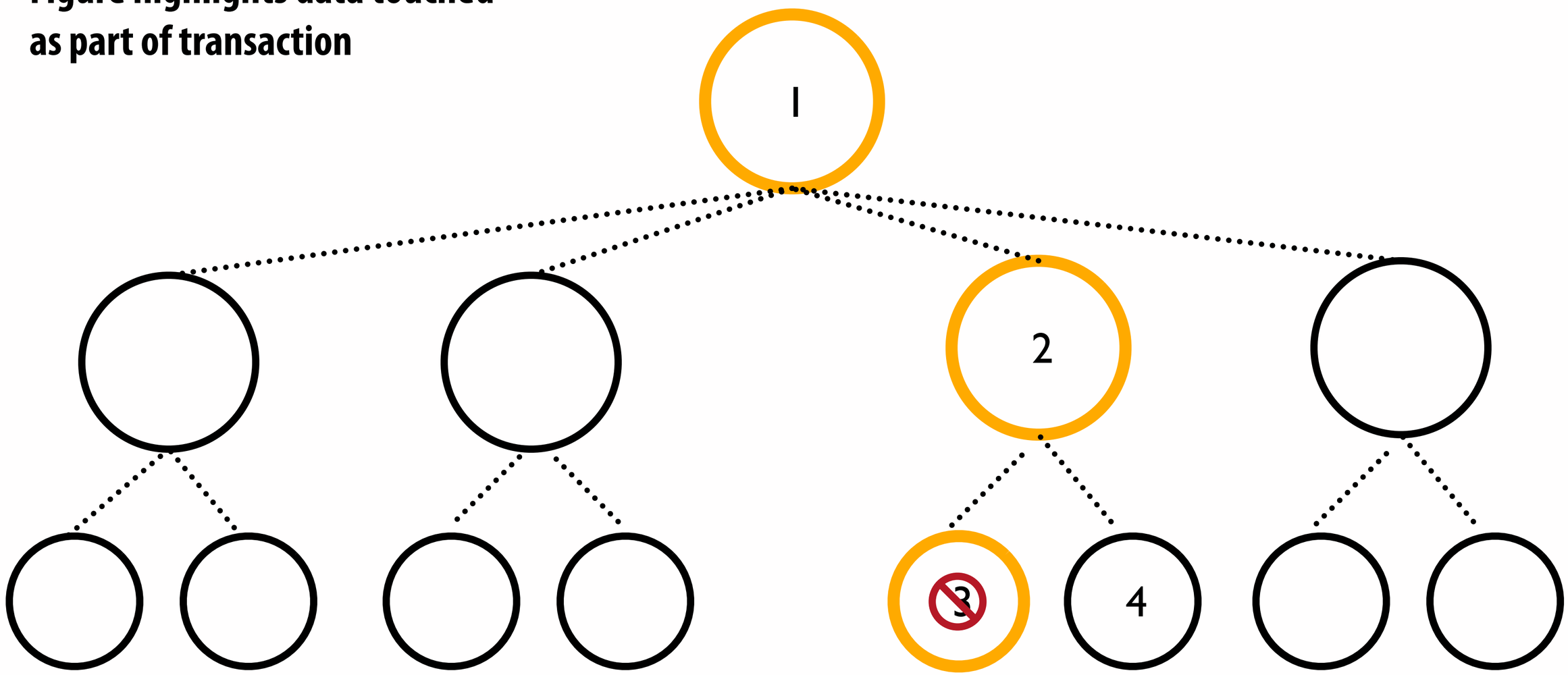


Transaction A

READ: 1, 2, 3

Transactions example

Figure highlights data touched as part of transaction



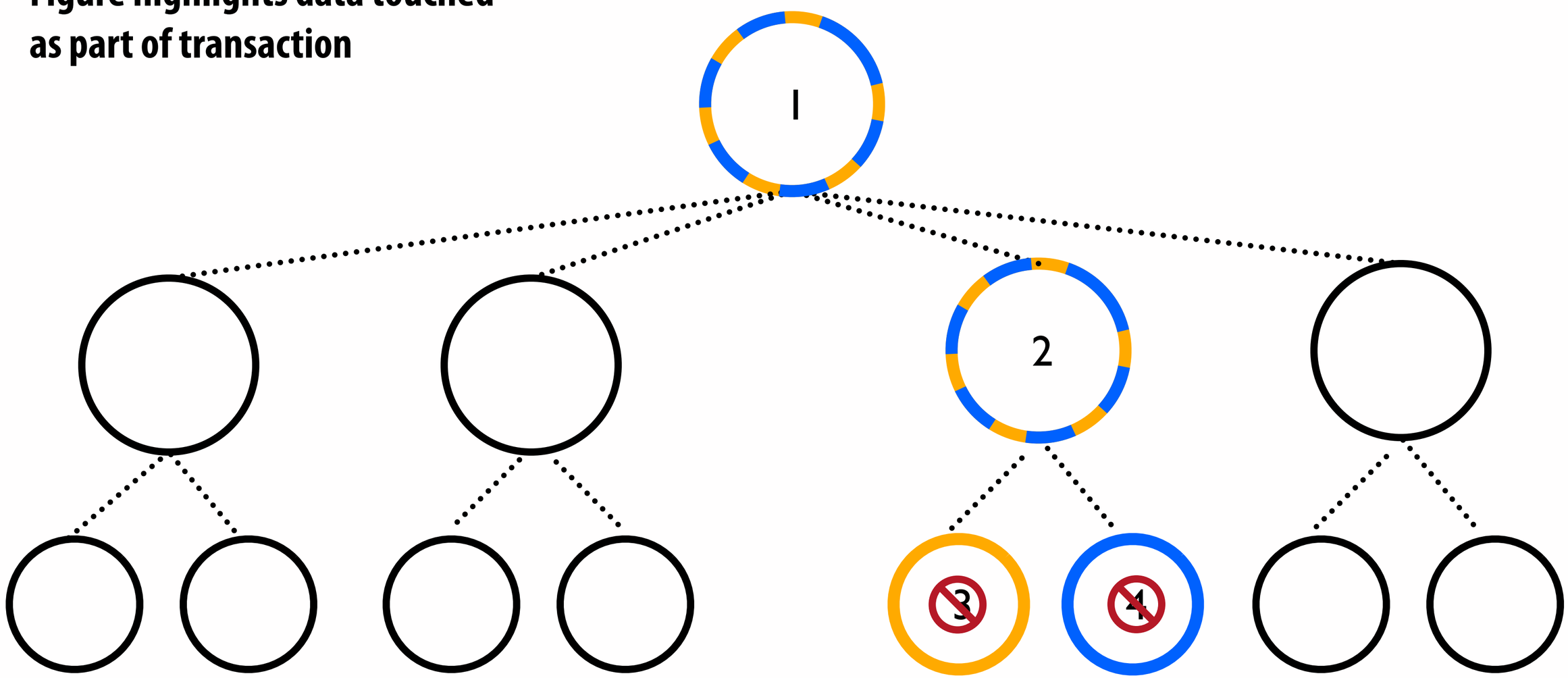
Transaction A

READ: 1, 2, 3

WRITE: 3

Transactions example

Figure highlights data touched as part of transaction



Transaction A
READ: 1, 2, 3
WRITE: 3

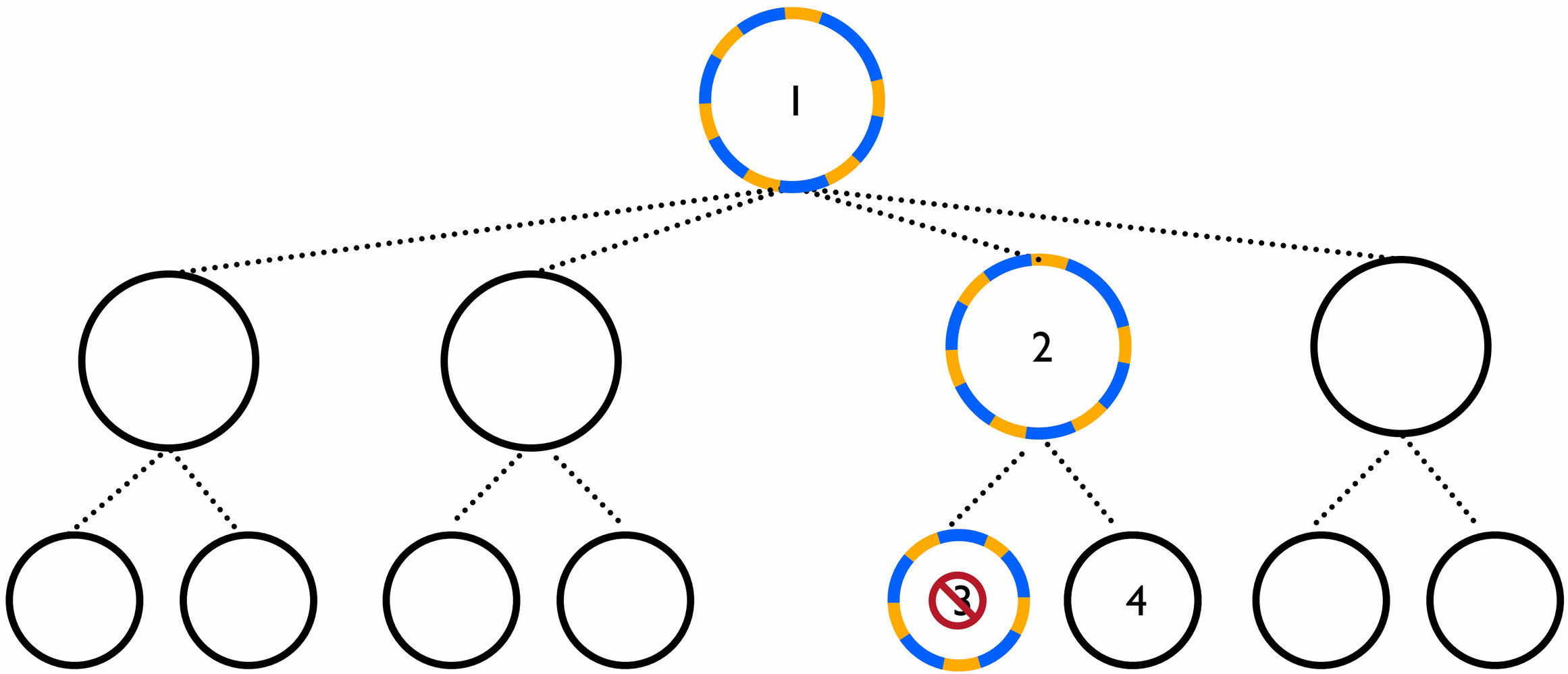
Transaction B
READ: 1, 2, 4
WRITE: 4

NO READ-WRITE or WRITE-WRITE conflicts!

(no transaction writes to data accessed by other transactions)

Transactions example 2

(Both transactions modify node 3)



Transaction A
READ: 1, 2, 3
WRITE: 3

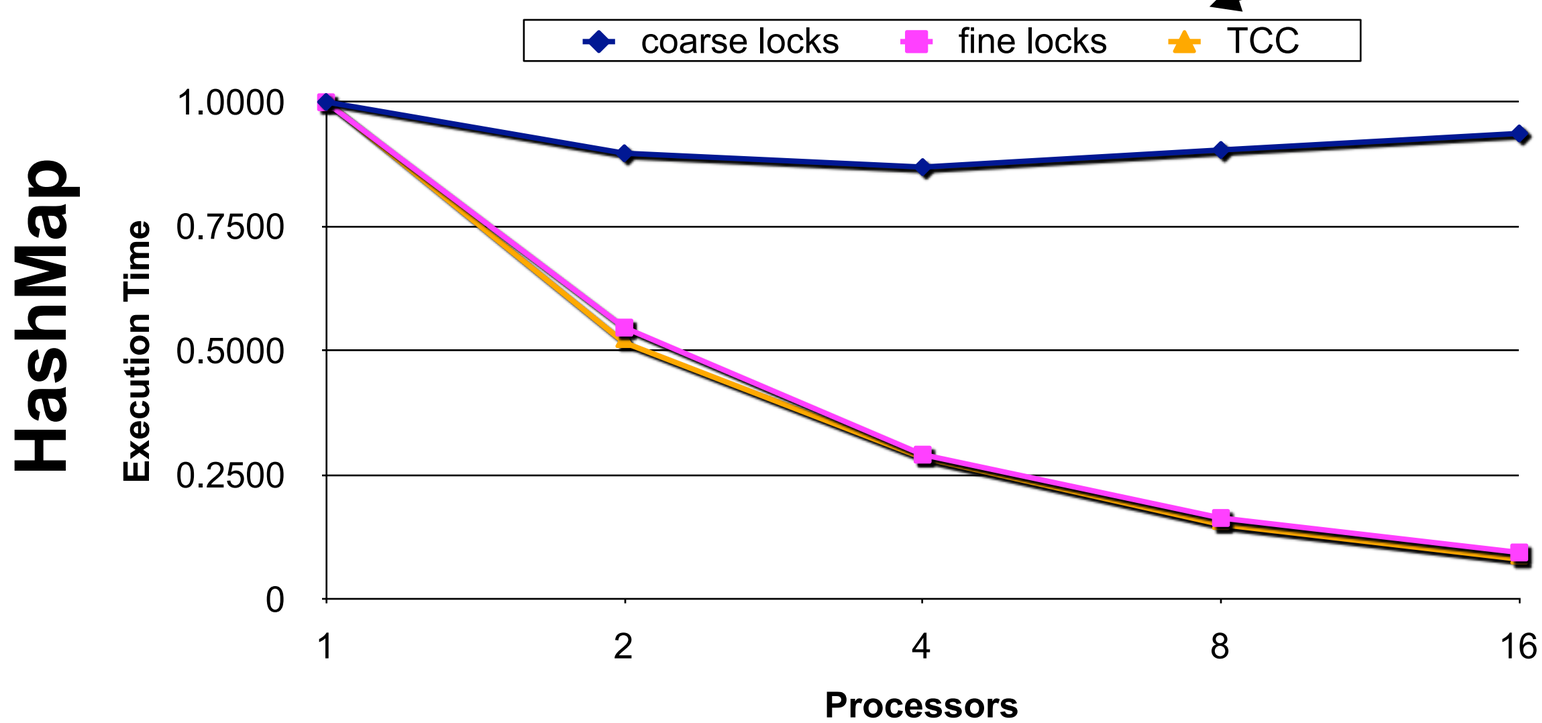
Transaction B
READ: 1, 2, 3
WRITE: 3

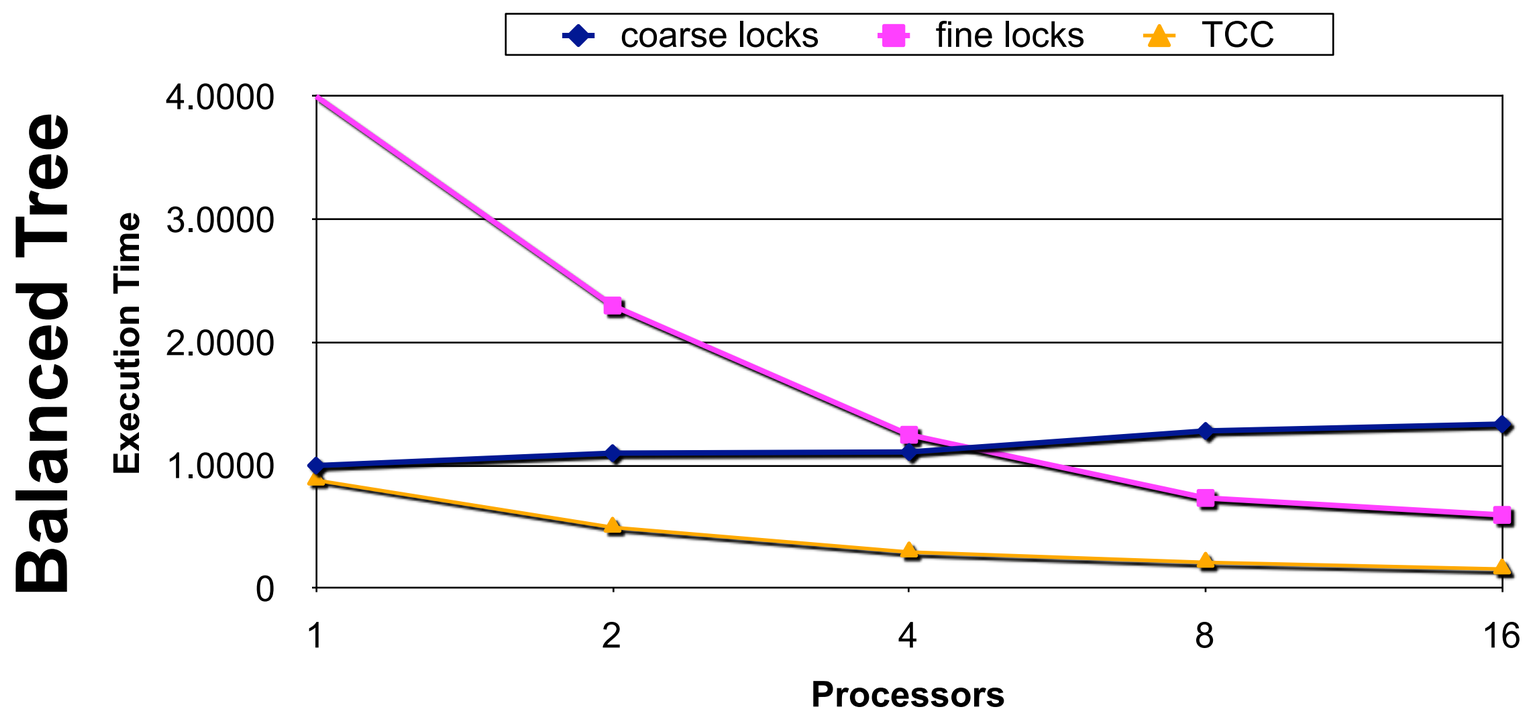
Conflicts exist: transactions must be serialized

(both transactions write to node 3)

Performance: locks vs. transactions

"TCC" is a HW-based TM system





Failure atomicity: locks

```
void transfer(A, B, amount) {
    synchronized(bank)
    {
        try {
            withdraw(A, amount);
            deposit(B, amount);
        }
        catch(exception1) { /* undo code 1*/ }
        catch(exception2) { /* undo code 2*/ }
        …
    }
}
```

- **Manually catch exceptions**
  - Programmer provides undo code on a case by case basis
  - Complexity: what to undo and how…
  - Some side-effects may become visible to other threads
    - E.g., an uncaught case can deadlock the system…

Failure atomicity: transactions

```
void transfer(A, B, amount)
{
    atomic {
        withdraw(A, amount);
        deposit(B, amount);
    }
}
```

- System processes exceptions
  - All but those explicitly managed by the programmer
  - Transaction is aborted and updates are undone
  - No partial updates are visible to other threads
    - E.g., no locks held by a failing threads…

Composability: locks

```
void transfer(A, B, amount)
{
   synchronized(A) {
     synchronized(B) {
       withdraw(A, amount);
       deposit(B, amount);
     }
   }
}
```

Thread 0:

`transfer(x,y)`

Thread 1:

`transfer(y,x);`

DEADLOCK!

- **Composing lock-based code can be tricky**
  - Requires system-wide policies to get correct
  - Breaks software modularity

- **Caught between an extra lock and a hard place**
  - Fine-grain locking: good for performance, but can lead to deadlock

Composability: locks

```
void transfer(A, B, amount) {
   synchronized(A) {
     synchronized(B) {
       withdraw(A, amount);
       deposit(B, amount);
     }
   }
}
```

DEADLOCK!

```
void transfer2(A, B, amount) {
  synchronized(B) {
    synchronized(A) {
      withdraw(A, 2*amount);
      deposit(B, 2*amount);
    }
  }
}
```

- **Composing lock-based code can be tricky**
  - Requires system-wide policies to get correct
  - Breaks software modularity

- **Caught between an extra lock and a hard place**
  - Fine-grain locking: good for performance, but can lead to deadlock

Composability: transactions

```
void transfer(A, B, amount) {
   atomic {
      withdraw(A, amount);
      deposit(B, amount);
   }
}
```

Thread 0:

```
transfer(x,y)
```

Thread 1:

```
transfer(y,x);
```

- Transactions compose gracefully
  - Programmer declares global intent (atomic transfer)
    - No need to know of global implementation strategy
  - Transaction in transfer subsumes those in withdraw & deposit
    - Outermost transaction defines atomicity boundary
- System manages concurrency as well as possible serialization
  - Serialization for transfer(A, B, 100) and transfer(B, A, 200)
  - Concurrency for transfer(A, B, 100) and transfer(C, D, 200)

Advantages of transactional memory

- **Easy to use synchronization construct**
  - As easy to use as coarse-grain locks
  - Programmer declares need for atomicity, system implements
- **Often performs as well as fine-grained locks**
  - Automatic read-read concurrency & fine-grain concurrency
- **Failure atomicity and recovery**
  - No lost locks when a thread fails
  - Failure recovery = transaction abort + restart
- **Composability**
  - Safe and scalable composition of software modules

Example integration with OpenMP

- **Example: OpenTM = OpenMP + TM**
  - OpenMP: master-slave parallel model
    - Easy to specify parallel loops & tasks
  - TM: atomic & isolation execution
    - Easy to specify synchronization and speculation

- **OpenTM features**
  - Transactions, transactional loops & sections
  - Data directives for TM (e.g., thread private data)
  - Runtime system hints for TM

- **Code example:**

```
#pragma omp transfor schedule (static, chunk=50)
for (int i=0; i<N; i++) {
    bin[A[i]]++;
}
```

Atomic { } ≠ lock() + unlock()

- **The difference**
  - Atomic: high-level declaration of atomicity
    - Does not specify implementation/blocking behavior
  - Lock: low-level blocking primitive
    - Does not provide atomicity or isolation on its own

- **Keep in mind**
  - Locks can be used to implement `atomic` block but…
  - Locks can be used for purposes beyond atomicity
    - Cannot replace all lock regions with atomic regions
  - `Atomic` eliminates many data races, but..
  - Programming with atomic blocks can still suffer from atomicity violations. e.g., programmer erroneous splits sequence that <u>should be</u> atomic into two atomic blocks

Example: lock-based code that does not work with atomic

```
// Thread 1
synchronized(lock1)
{
  …
  flagA = true;
  while (flagB == 0);
  …
}
```

```
// Thread 2
synchronized(lock2)
{
  …
  flagB = true;
  while (flagA == 0);
  …
}
```

- **What is the problem with replacing synchronized with atomic?**

Example: atomicity violation

```
// Thread 1
atomic()
{
  …
  ptr = A;
  …
}

atomic() {
  B = ptr->field;
}
```

```
// Thread 2
atomic
{
  …
  ptr = NULL;
}
```

- **Programmer mistake: logically atomic code sequence (in thread 1) is separated into two atomic blocks.**

Transactional memory: summary + benefits

- TM = declarative synchronization
  - User specifies requirement (atomicity and isolation)
  - System implements semantics in best possible way
- Motivation for TM
  - Difficult for user to get explicit synchronization right
    - Correctness vs. performance vs. complexity
  - Explicit synchronization is difficult to scale
    - Locking scheme for four CPUs is often not the best scheme for 64 CPUs
  - Explicit synchronization can break composability of software
    - Need a globally-adhered to locking policy
  - Other advantages: fault atomicity, …
- Productivity argument:
  - System support for transactions can achieve 90% of the benefit of programming with fined-grained locks, with 10% of the development time

Implementing transactional memory

Recall: transactional memory

- **Atomicity (all or nothing)**
  - At commit, all memory writes take effect at once
  - On abort, none of the writes appear to take effect
- **Isolation**
  - No other code can observe writes before commit
- **Serializability**
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed though

TM implementation basics

- TM systems must provide atomicity and isolation
  - Without sacrificing concurrency
- Basic implementation requirements
  - Data versioning (ALLOWS abort)
  - Conflict detection & resolution (WHEN to abort)
- Implementation options
  - Hardware transactional memory (HTM)
  - Software transactional memory (STM)
  - Hybrid transactional memory
    - e.g., hardware-accelerated STMs

Data versioning

Manage uncommited (new) and commited (old) versions of data for concurrent transactions

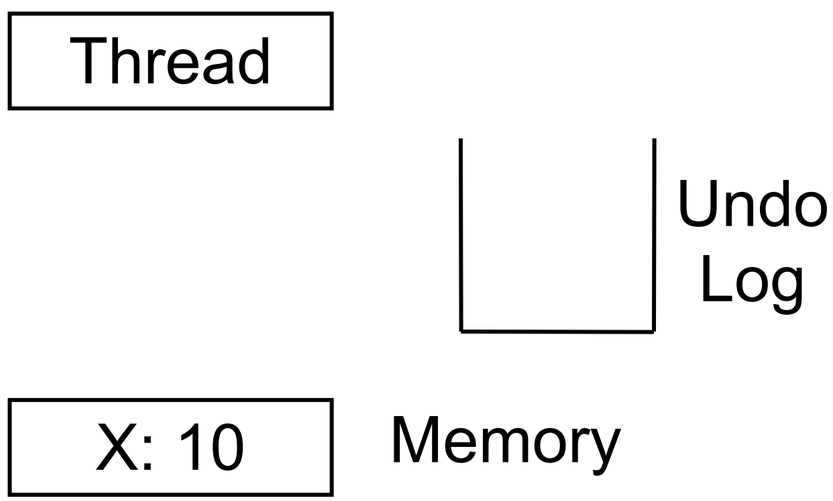
1. **Eager versioning (undo-log based)**

2. **Lazy versioning (write-buffer based)**

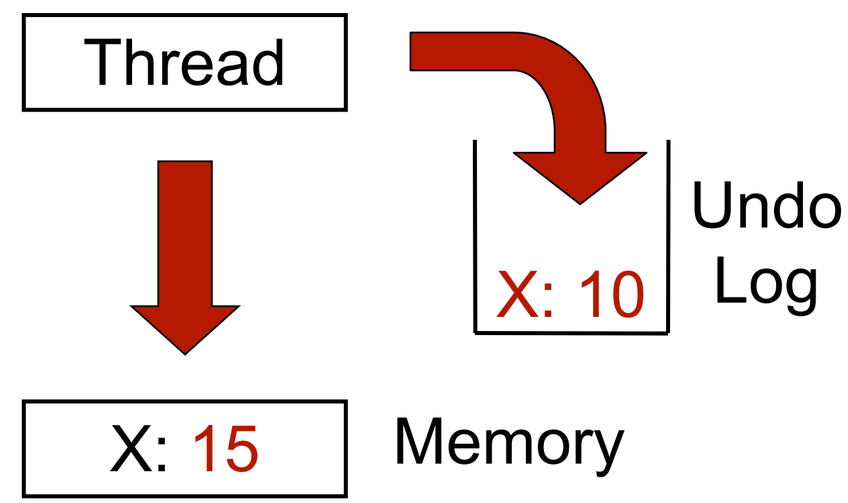
Eager versioning

Update memory immediately, maintain "undo log" in case of abort

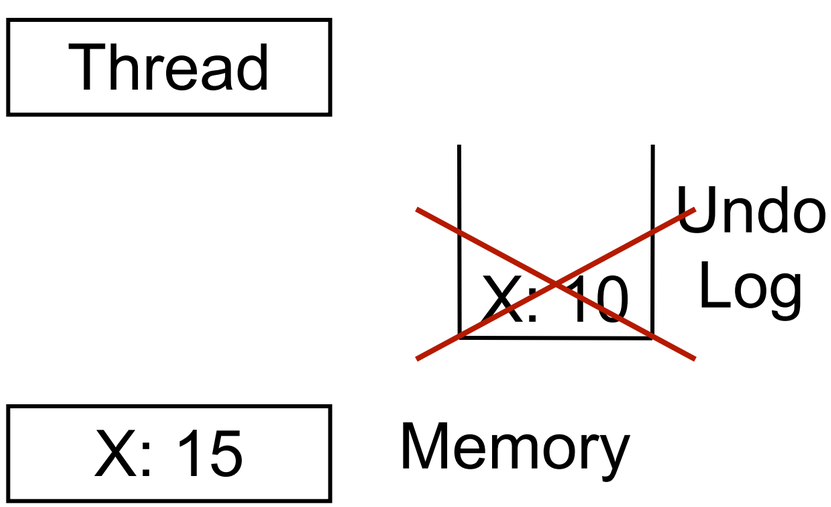
Begin Xaction



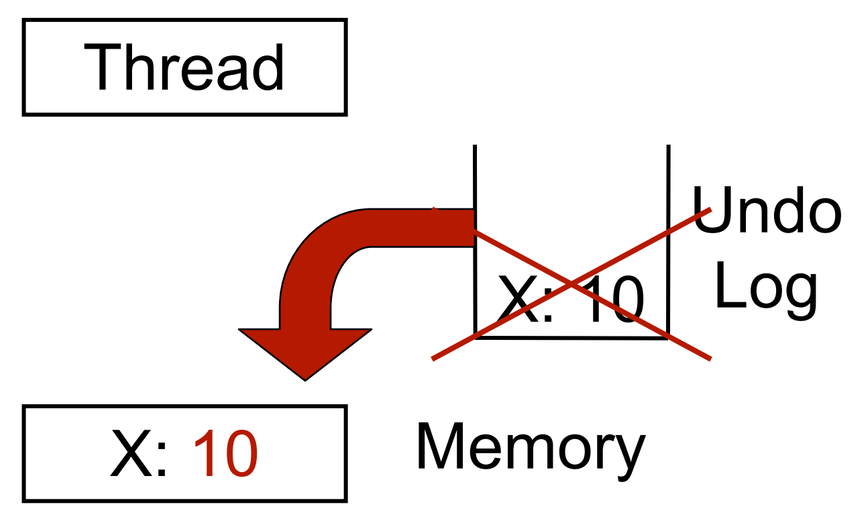
Write X←15



Commit Xaction

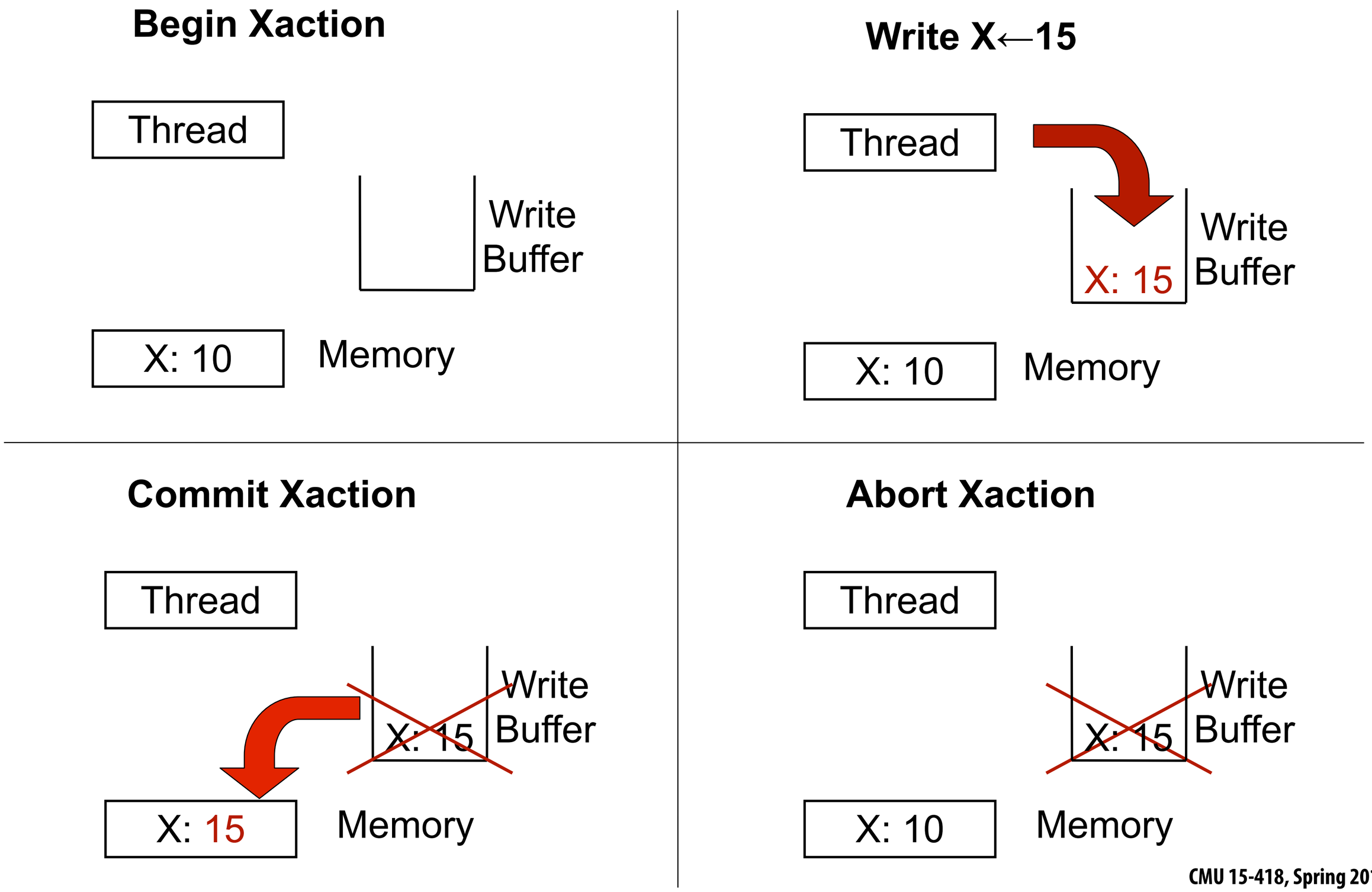


Abort Xaction



Lazy versioning

Log memory updates in transaction write buffer, flush buffer on commit



Data versioning

- Manage uncommitted (new) and committed (old) versions of data for concurrent transactions
- Eager versioning (undo-log based)
  - Update memory location directly
  - Maintain undo info in a log (per store overhead)
  - Good: faster commit (data is already in memory)
  - Bad: slower aborts, fault tolerance issues (crash in middle of trans)
- Lazy versioning (write-buffer based)
  - Buffer data in a write buffer until commit
  - Update actual memory location on commit
  - Good: faster abort (clear log), no fault tolerance issues
  - Bad: Slower commits

Conflict detection

- Detect and handle conflicts between transactions
  - Read-write conflict: transaction A reads address X, which was written to by pending transaction B
  - Write-write conflict: transactions A and B are pending, both write to address X.

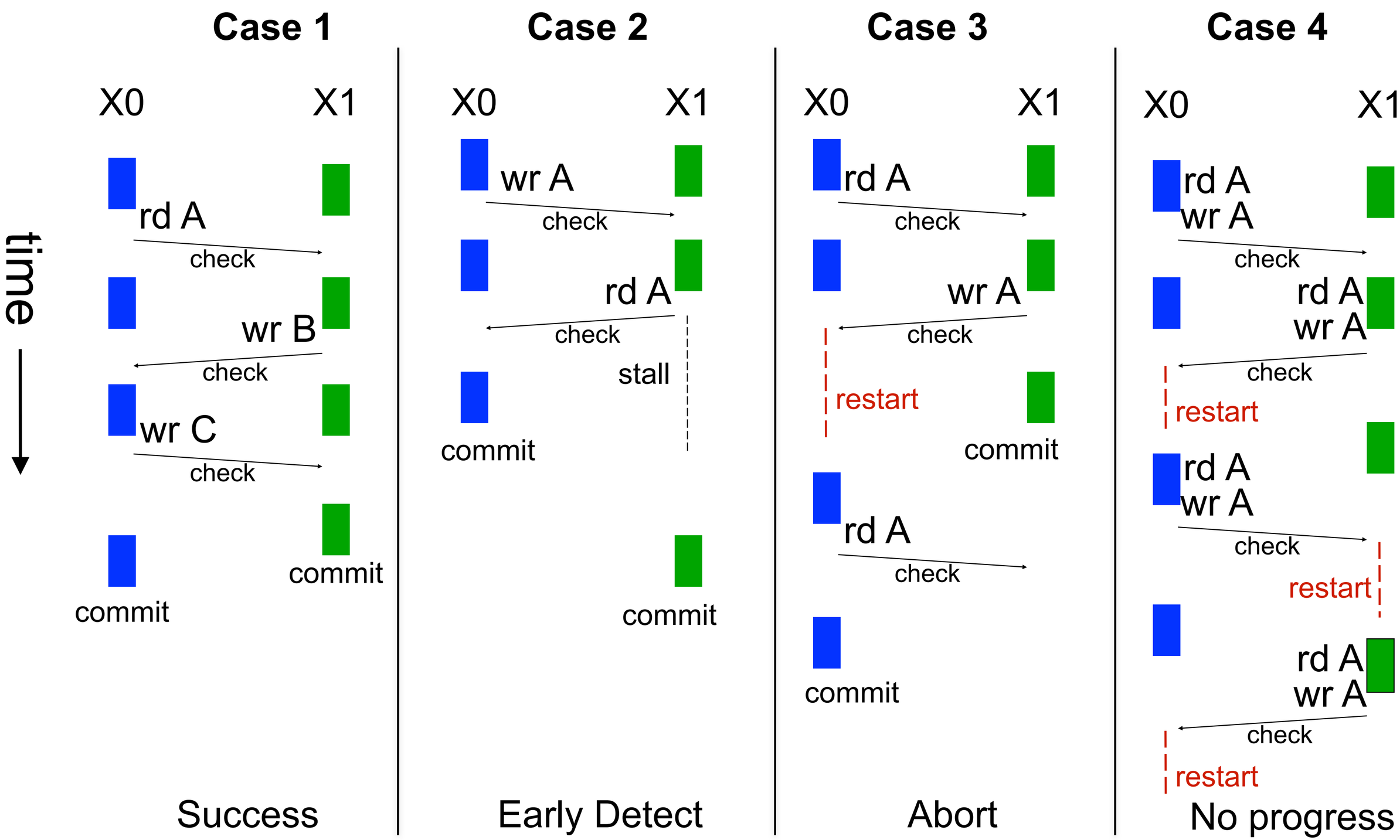
- System must track a transaction's read set and write set
  - Read-set: addresses read within the transaction
  - Write-set: addresses written within transaction

Pessimistic detection

- Check for conflicts during loads or stores
  - e.g., HW implementation will check through coherence actions (will discuss later)

- "Contention manager" decides to stall or abort transaction
  - Various priority policies to handle common case fast

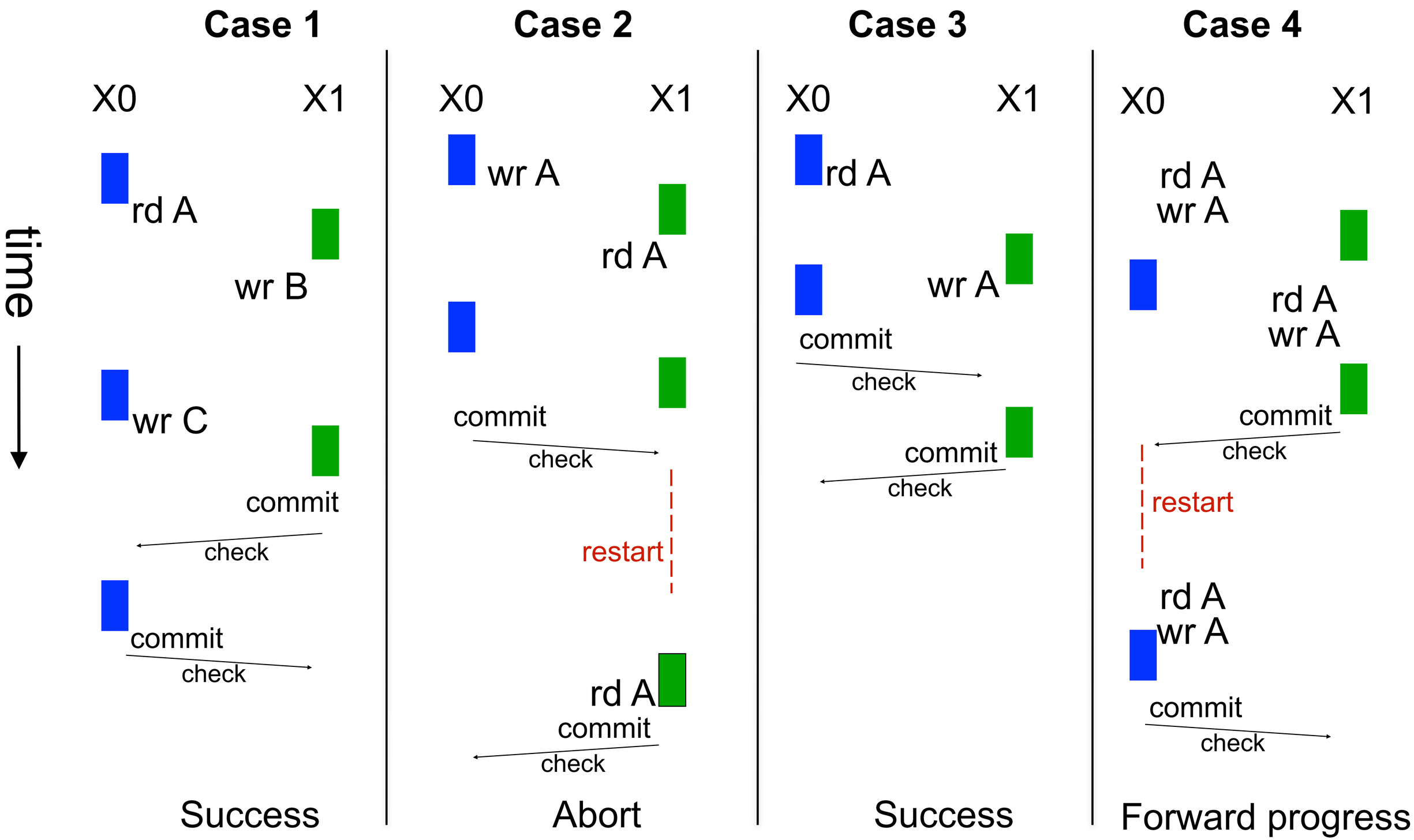
Pessimistic detection example



Optimistic detection

- Detect conflicts when a transaction attempts to commit
  - HW: validate write set using coherence actions
    - Get exclusive access for cache lines in write set
- On a conflict, give priority to committing transaction
  - Other transactions may abort later on
  - On conflicts between committing transactions, use contention manager to decide priority
- Note: can use optimistic and pessimistic schemes together
  - Several STM systems use optimistic for reads and pessimistic for writes

Optimistic detection



Conflict detection trade-offs

- Pessimistic conflict detection (a.k.a. "encounter" or "eager")
  - Good: Detect conflicts early (undo less work, turn some aborts to stalls)
  - Bad: no forward progress guarantees, more aborts in some cases
  - Bad: fine-grained communication
  - Bad: detection on critical path

- Optimistic conflict detection (a.k.a. "commit" or "lazy")
  - Good: forward progress guarantees
  - Good: potentially less conflicts, bulk communication
  - Bad: detects conflicts late, can still have fairness problems

Conflict detection granularity

- **Object granularity (SW-based techniques)**
  - Good: reduced overhead (time/space)
  - Good: close to programmer's reasoning
  - Base: false sharing on large objects (e.g. arrays)
- **Machine word granularity**
  - Good: minimize false sharing
  - Bad: increased overhead (time/space)
- **Cache-line granularity**
  - Good: compromise between object & word
- **Can mix & match to get best of both words**
  - Word-level for arrays, object-level for other data, …

TM implementation space (examples)

- **Hardware TM systems**
  - Lazy + optimistic: Stanford TCC
  - Lazy + pessimistic: MIT LTM, Intel VTM
  - Eager + pessimistic: Wisconsin LogTM
  - Eager + optimistic: not practical
- **Software TM systems**
  - Lazy + optimistic (rd/wr): Sun TL2
  - Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
  - Eager + optimistic (rd)/pessimistic (wr): Intel STM
  - Eager + pessimistic (rd/wr): Intel STM
- **Optimal design remains an open question**
  - May be different for HW, SW, and hybrid

Hardware transactional memory (HTM)

- **Data versioning in caches**
  - Cache the write buffer or the undo log
  - Add new cache line metadata to track read set and write set
  - Can do with private, shared, and multi-level caches

- **Conflict detection through cache coherence protocol**
  - Coherence lookups detect conflicts between transactions
  - Works with snooping and directory coherence

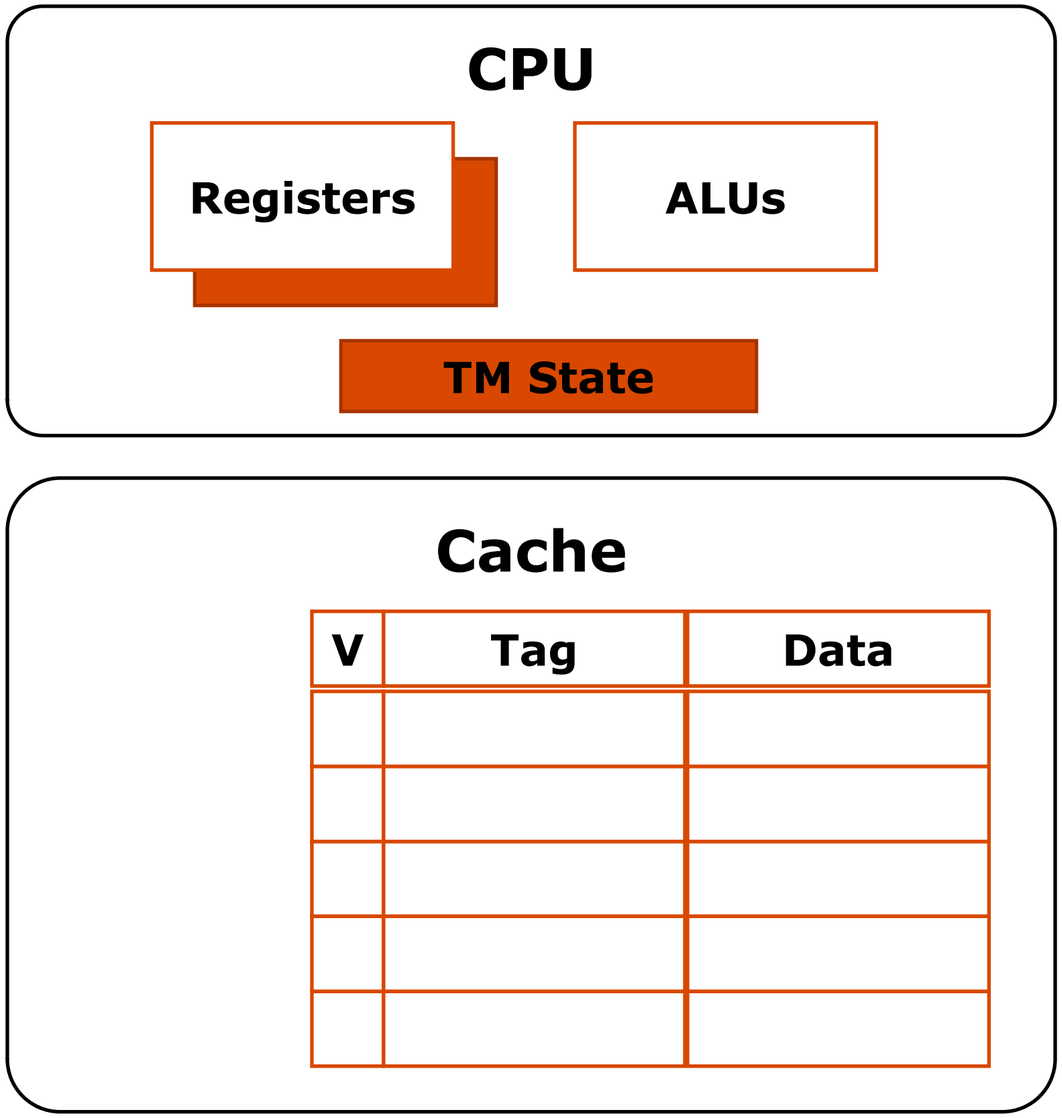
- **Notes**
  - Register checkpoint must be taken at transaction begin

HTM design

- **Cache lines annotated to track read set and write set**
  - R bit: indicates data read by transaction; set on loads
  - W bit: indicates data written by transaction; set on stores
    - R/W bits can be at word or cache-line granularity
  - R/W bits gang-cleared on transaction commit or abort
  - For eager versioning, need a 2nd cache write for undo log

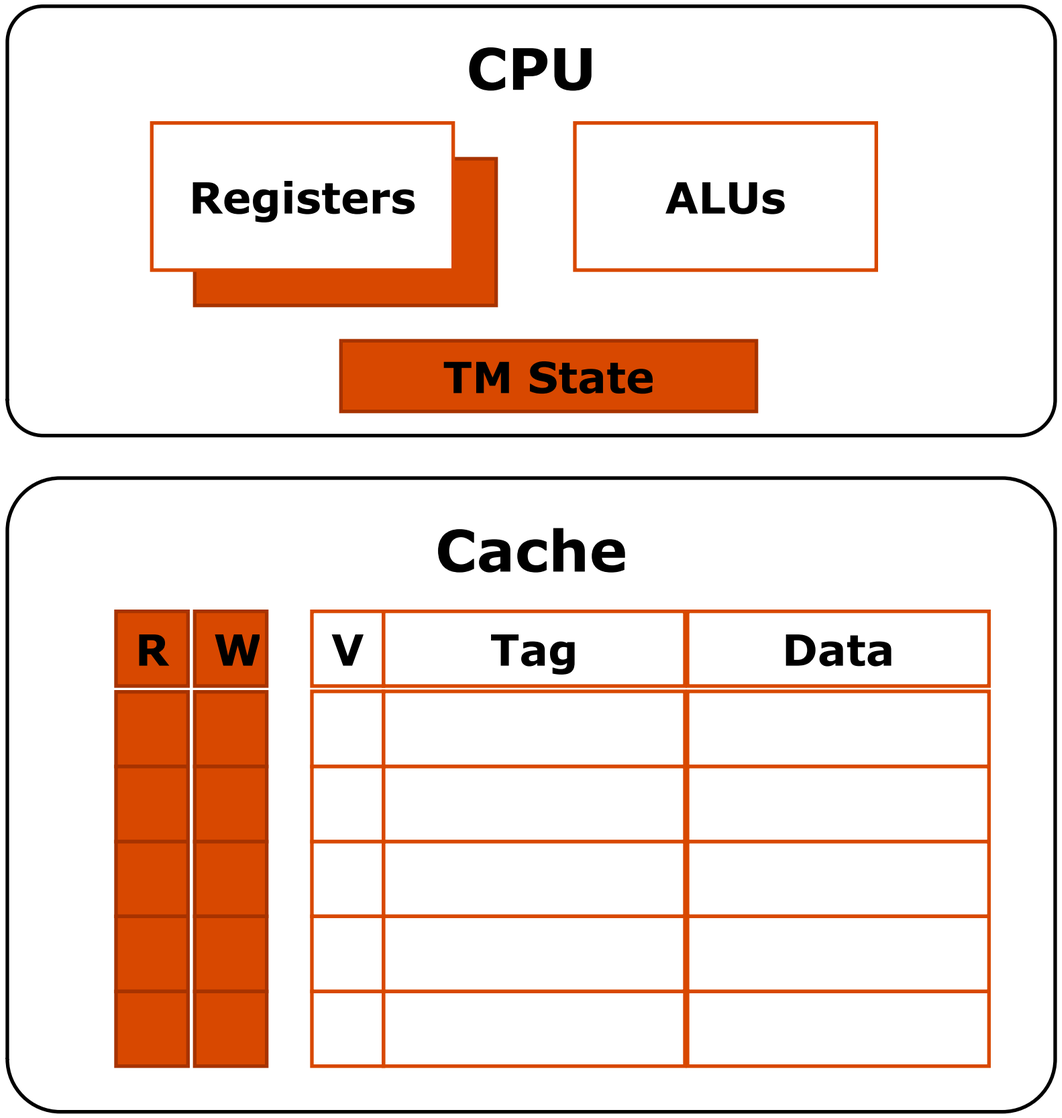
- **Coherence requests check R/W bits to detect conflicts**
  - Shared request to W-word is a read-write conflict
  - Exclusive request to R-word is a write-read conflict
  - Exclusive request to W-word is a write-write conflict

Example HTM: lazy optimistic



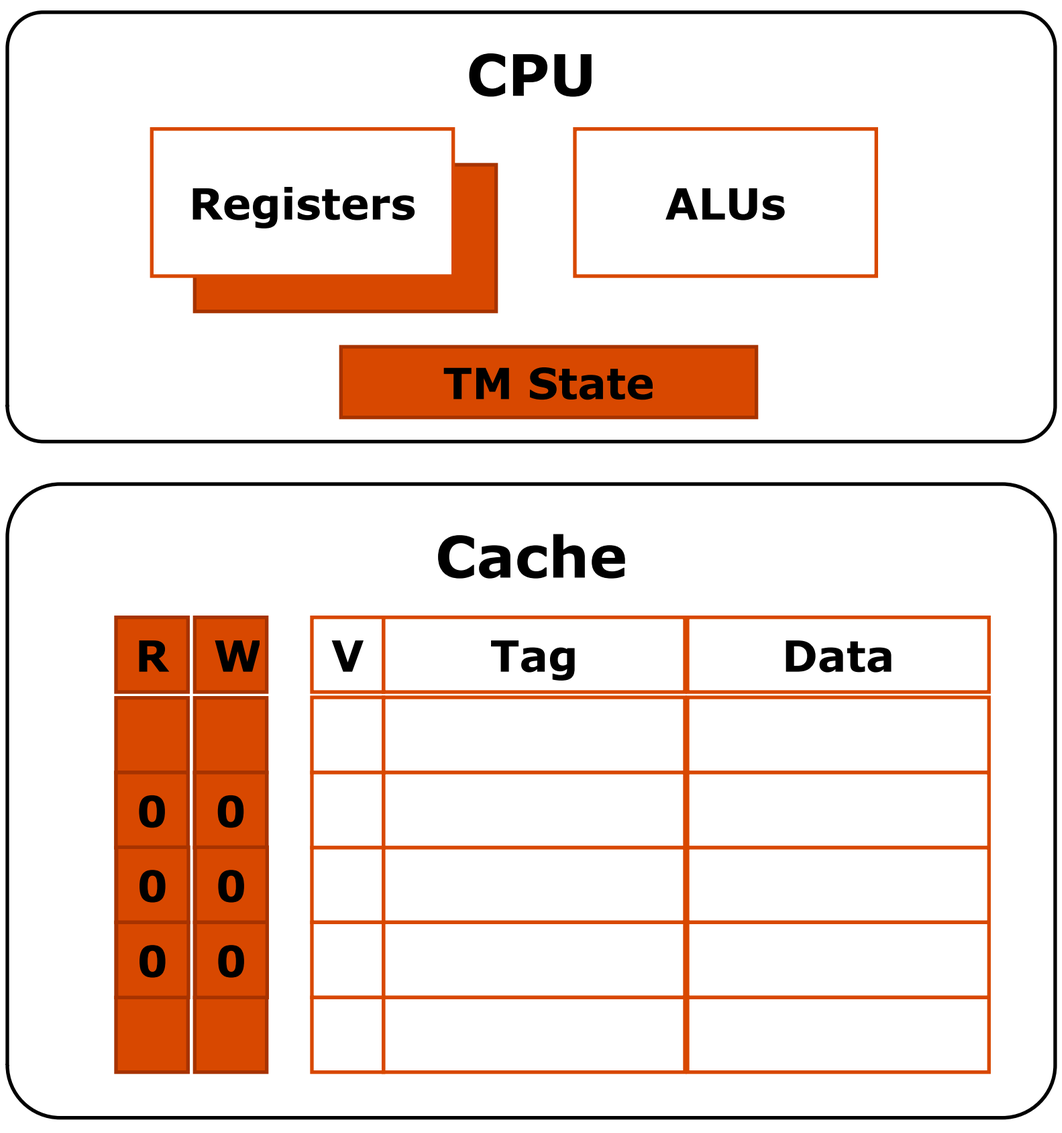
- **CPU changes**
  - Register checkpoint (available in many CPUs)
  - TM state registers (status, pointers to handlers, …)

Example HTM: Lazy Optimistic



- **Cache changes**
  - R bit indicates membership to read set
  - W bit indicates membership to write set

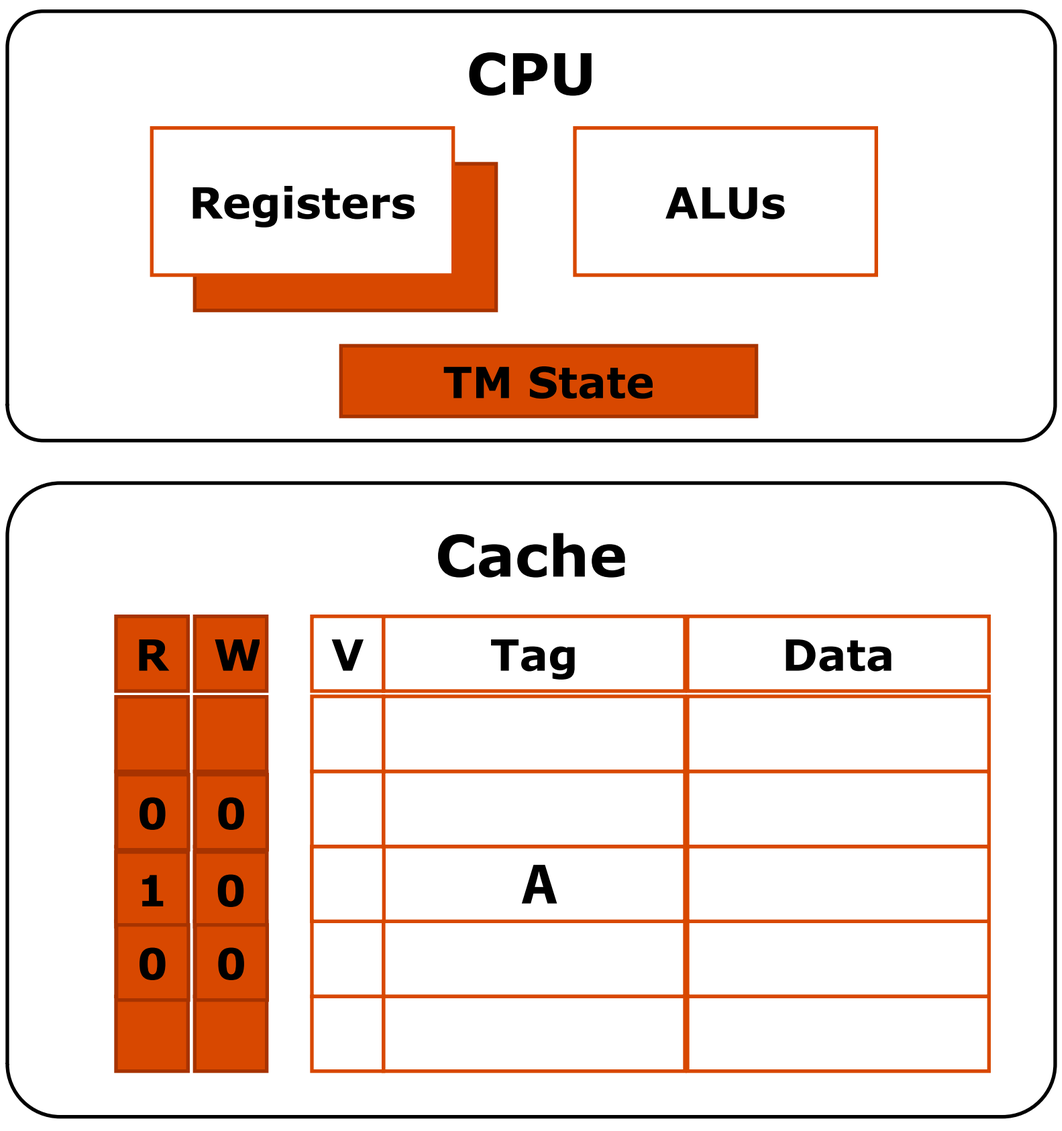
HTM transaction execution




```
Xbegin ⬅
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

- **Transaction begin**
  - Initialize CPU & cache state
  - Take register checkpoint

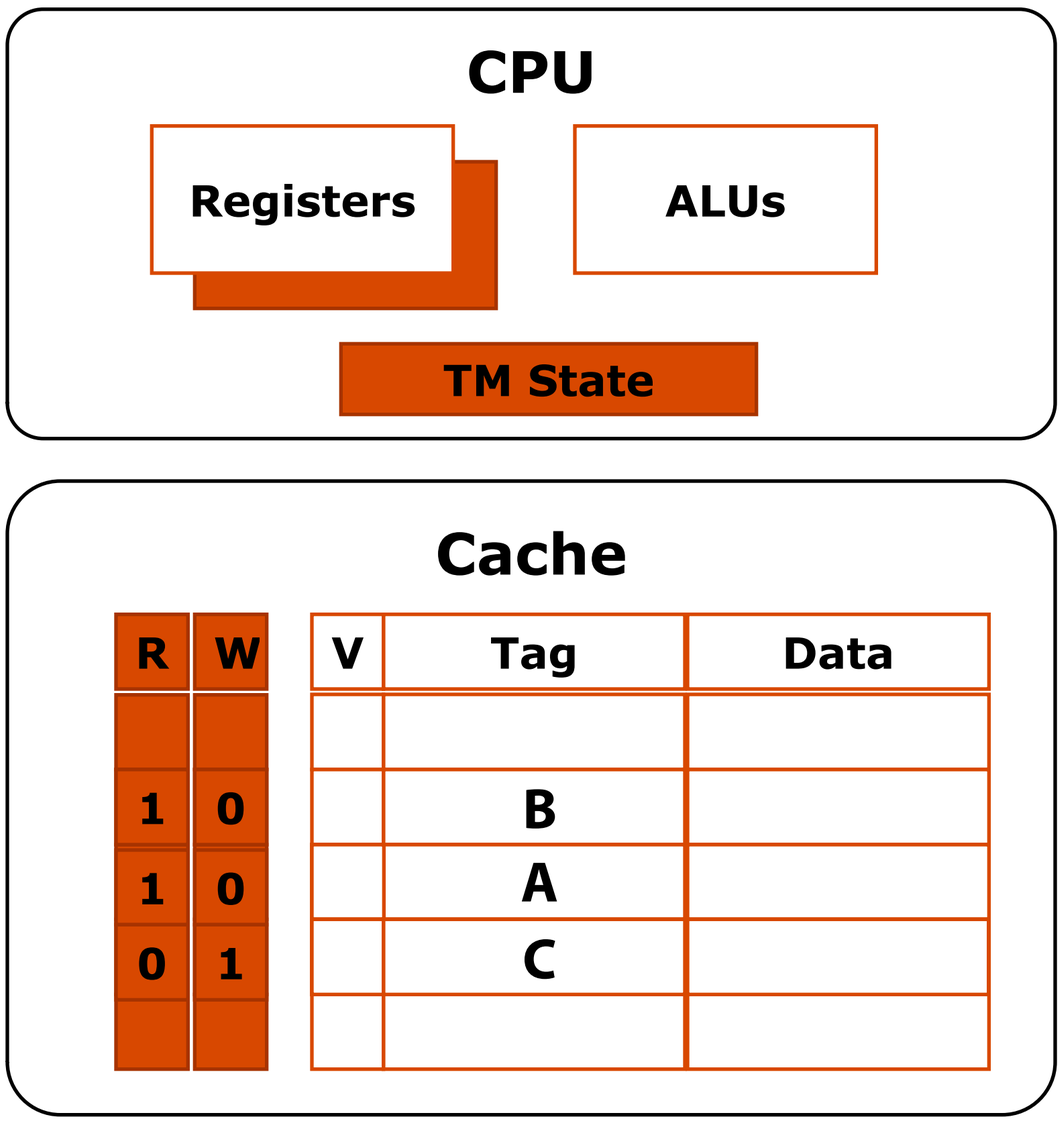
HTM transaction execution




```
Xbegin
    Load A  ←
    Load B
    Store C ⇐ 5
Xcommit
```

- **Load operation**
  - Serve cache miss if needed
  - Mark data as part of read set

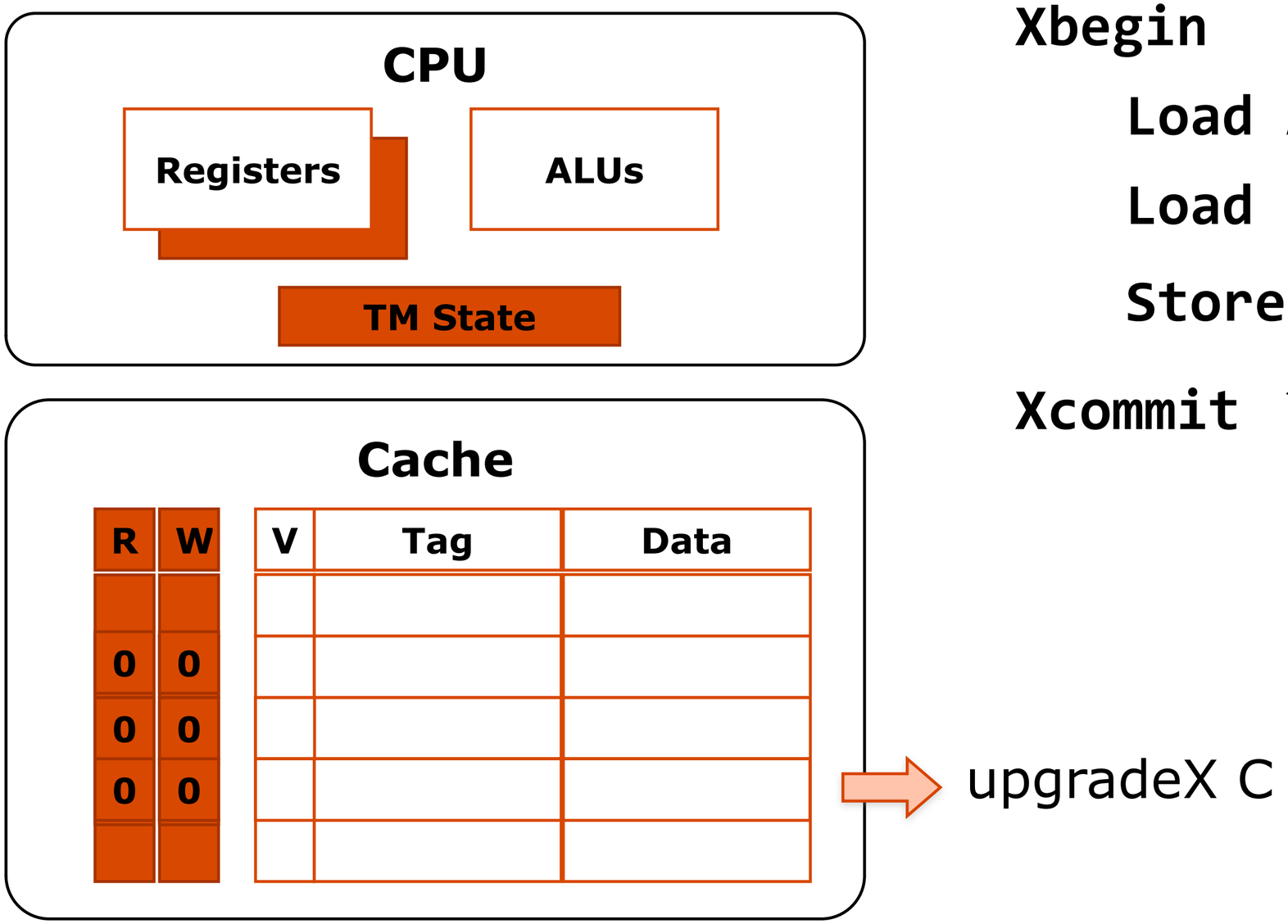
HTM transaction execution




```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

- **Store operation**
  - Server cache miss if needed
  - Mark data as part of write set

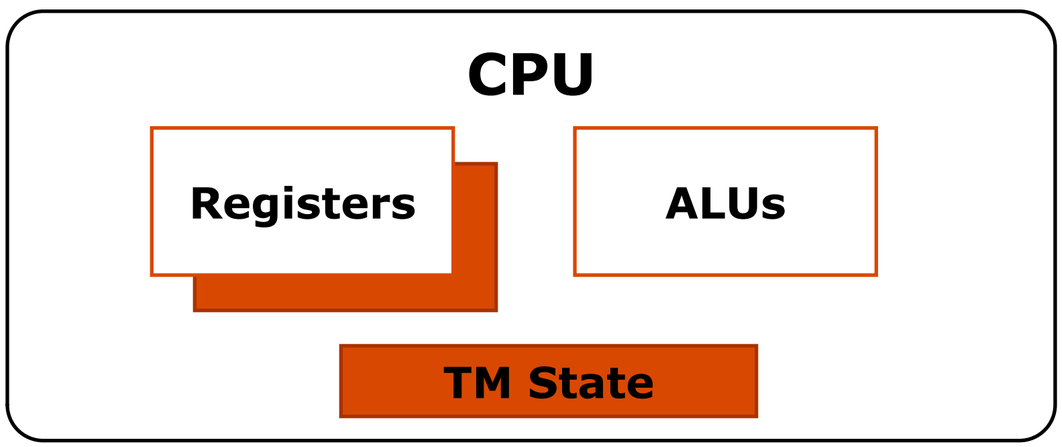
HTM transaction execution: commit



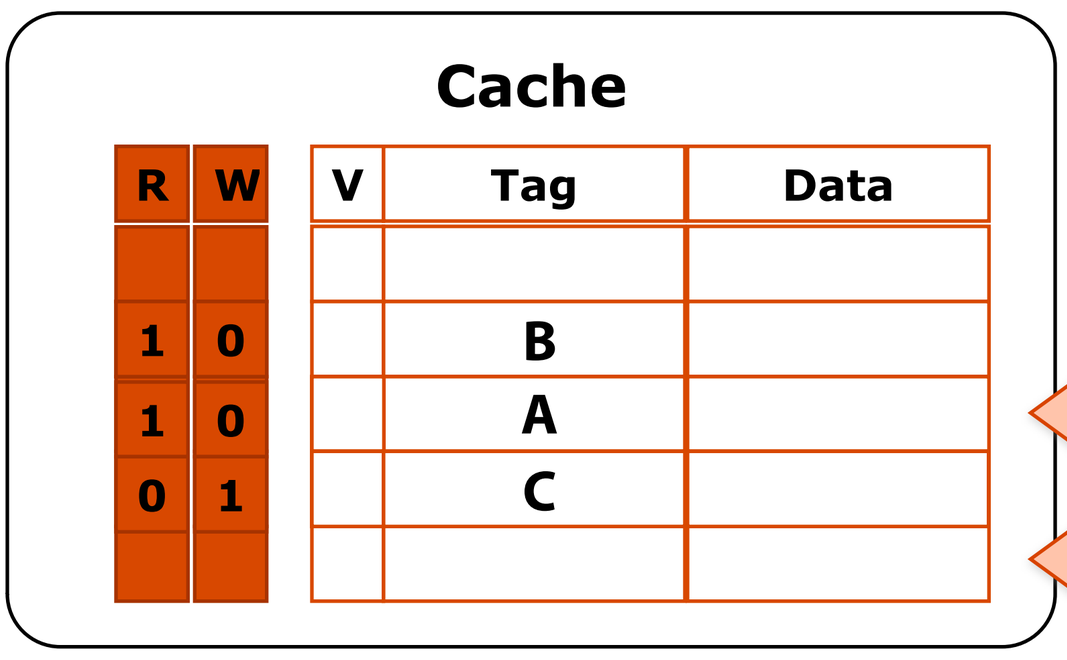

```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit ⬅
```

- **Fast two-phase commit**
  - Validate: request exclusive access to write set lines (if needed)
  - Commit: gang-reset R and W bits, turns write set data to valid (dirty) data

HTM transaction execution: detect/abort




```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```



upgradeX A ✗

upgradeX D ✓

coherence requests from another core's commit

(remote core's write of A conflicts with local read of A: triggers abort of pending local transaction)

- **Fast conflict detection and abort**
  - Check: lookup exclusive requests in the read set and write set
  - Abort: invalidate write set, gang-reset R and W bits, restore checkpoint

Transactional memory support in Intel's upcoming Haswell architecture

- New instructions for "restricted transactional memory" (RTM)
  - `xbegin:` takes pointer to "fallback address" in case of abort
  - `xend`
  - `xabort`

- Processor makes sure all memory operations commit atomically
  - But processor may automatically abort transaction for many reasons
  - Programmer's manual gives guidelines for increasing probability transactions will not abort

Transactional memory summary

- **Atomic construct: declaration of atomic behavior**
  - Motivating idea: increase simplicity of synchronization, without sacrificing performance
- **Transactional memory implementation**
  - Many variants have been proposes: SW, HW, SW+HW
  - Differ in versioning policy (eager vs. lazy)
  - Conflict detection policy (pessimistic vs. optimistic)
  - Detection granularity
- **Hardware transactional memory**
  - Versioned data kept in caches
  - Conflict detection built upon coherence protocol