Lecture 19:

# Transactional Memory

---

## Parallel Computer Architecture and Programming
## CMU 15-418/15-618, Spring 2018

Credit: many of the slides in today's talk are borrowed from Professor Christos Kozyrakis (Stanford University, now EPFL)

# Raising level of abstraction for synchronization

- **Previous topic: machine-level atomic operations**
  - Fetch-and-op, test-and-set, compare-and-swap, load linked-store conditional

- **Then we used these atomic operations to construct higher level synchronization primitives in software:**
  - Locks, barriers
  - We've seen how it can be challenging to produce correct programs using these primitives (easy to create bugs that violate atomicity, create deadlock, etc.)

- **Today: raising level of abstraction for synchronization even further**
  - Idea: transactional memory

# What you should know

- What a transaction is

- The difference (in semantics) between an `atomic` code block and `lock/unlock` primitives
- The basic design space of transactional memory implementations
  - Data versioning policy
  - Conflict detection policy
  - Granularity of detection

- The basics of a hardware implementation of transactional memory (consider how it relates to the cache 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
- Locks are one mechanism to synchronize threads to ensure atomicity of update (via ensuring 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 use or management of locks

- **System implements synchronization as necessary to ensure atomicity**
  - System could implement atomic { } using a lock
  - Implementation discussed today uses optimistic concurrency: serialization only in situations of 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 a shared task queue
  - Acquire a lock, perform operations, release the lock

# Transactional Memory (TM)

- **Memory transaction**
  - An atomic and isolated sequence of memory accesses
  - Inspired by database transactions
- **Atomicity (all or nothing)**
  - Upon transaction commit, all memory writes in transaction take effect at once
  - On transaction abort, none of the writes appear to take effect (as if transaction never happened)
- **Isolation**
  - No other processor can observe writes before transaction commits
- **Serializability**
  - Transactions appear to commit in a single serial order
  - But the exact order of commits is not guaranteed by semantics of transaction

# Transactional Memory (TM)

- In other words… many of the properties we maintained for a single address in a coherent memory system, we'd like to maintain for sets of reads and writes in a transaction.

**Transaction:**

  **Reads: X, Y, Z**

  **Writes: A, X**

These memory transactions will either all be observed by other processors, or none of them will. (they effectively all happen at the same time)

# Load-linked, store conditional (LL/SC)

- LL/SC is a light version of transactional memory
- Pair of corresponding instructions (not a single atomic instruction like compare-and-swap)
  - load_linked(x): load value from address
  - store_conditional(x, value): store value to x, if x hasn't been written to since corresponding LL

- Corresponding ARM instructions: LDREX and STREX
- How might LL/SC be implemented on a cache coherent processor?

# Motivating transactional memory

# Another example: Java 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 (when synchronization needed)**

**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 (myHashMap) {   // guards all accesses to hashMap
    return myHashMap.get(key);
  }
}
```

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

# Review from earlier fine-grained sync lecture

## What are solutions for making Java's HashMap thread-safe?

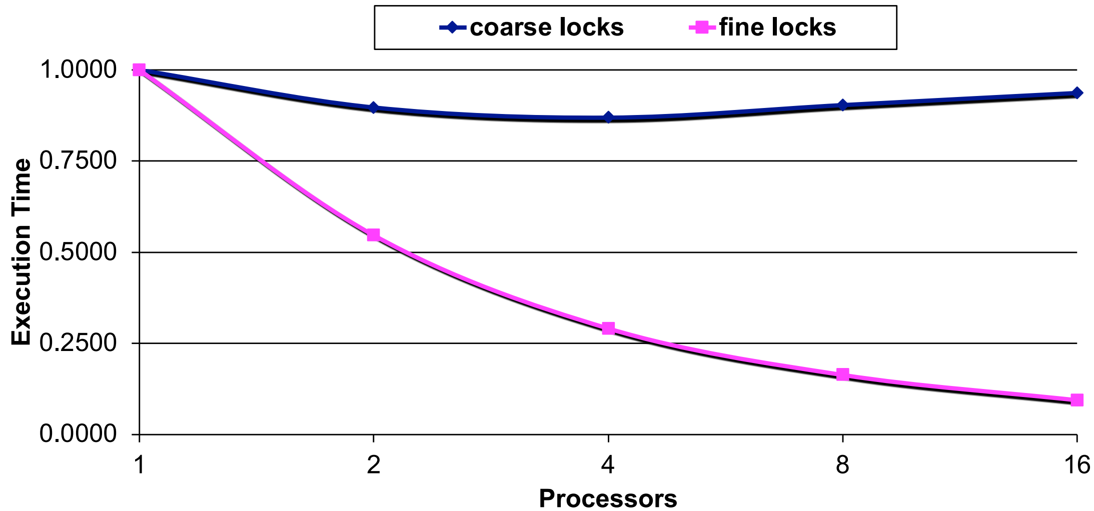
```
public Object get(Object key)  {
  int idx = hash(key);              // compute hash
  HashEntry e = buckets[idx];       // find bucket
  while (e != null)  {              // find element in bucket
    if (equals(key, e.key))
      return e.value;
    e = e.next;
  }
  return null;
}
```

- **One solution: use finer-grained synchronization (e.g., lock per bucket)**
  - **Now thread safe: but incurs lock overhead even if synchronization not needed**

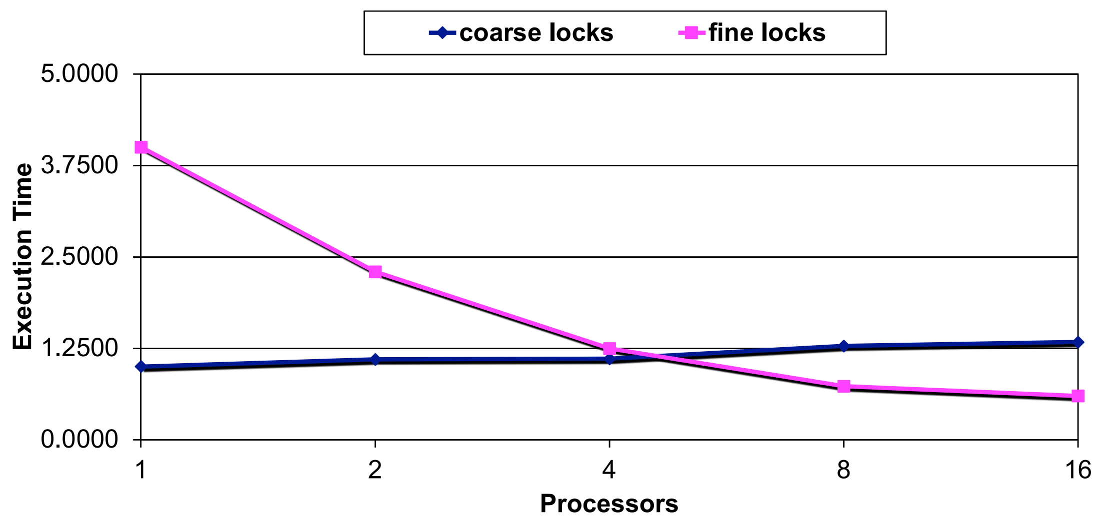
# Review: performance of fine-grained locking

## Reducing contention via fine-grained locking leads to better performance

Hash-Table




Balanced Tree



# Transactional HashMap

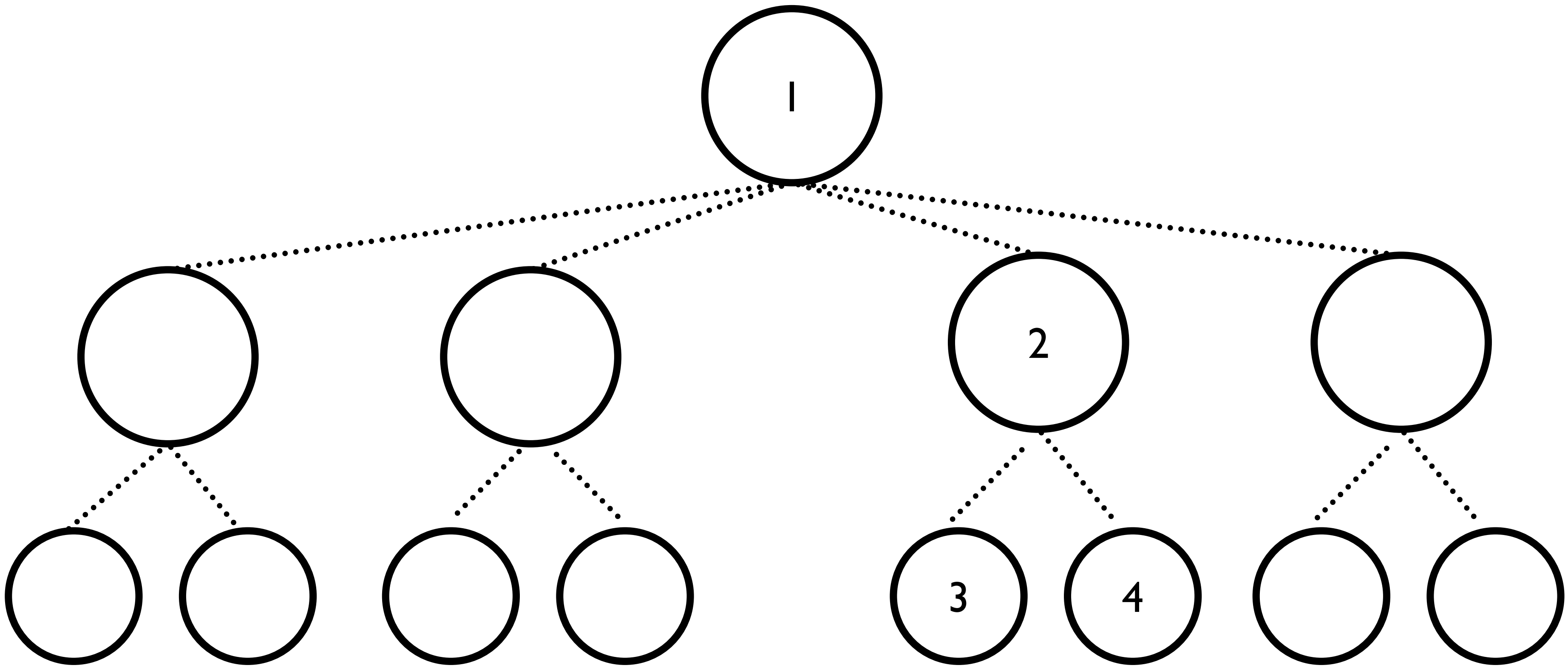
- **Simply enclose all operation in atomic block**
  - **Semantics of atomic block: system ensures atomicity of logic within block**

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

- **Transactional HashMap**
  - **Good: thread-safe, easy to program**
  - **What about performance and scalability?**
    - **Depends on the workload and implementation of atomic (to be discussed)**

# Another example: tree update by two threads

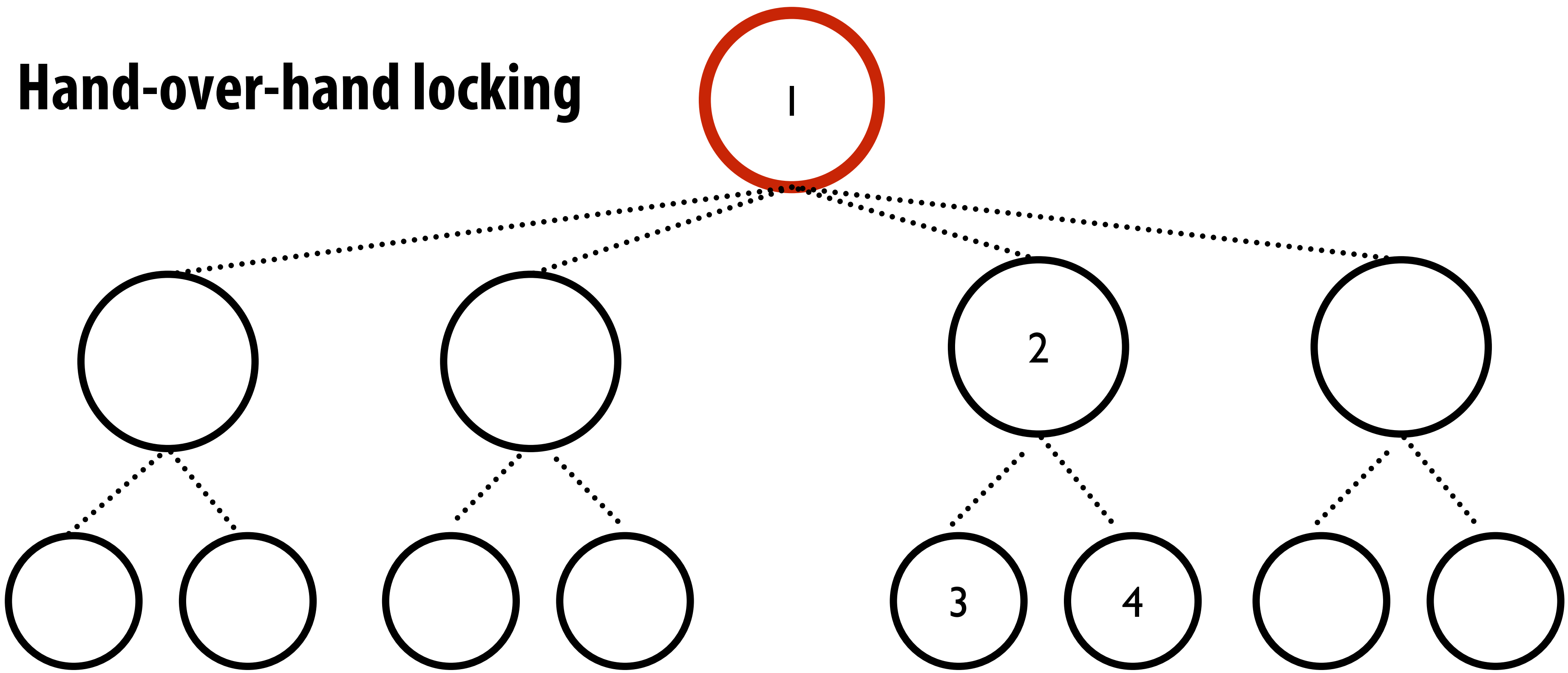
## Goal: modify nodes 3 and 4 in a thread-safe way



# Fine-grained locking example

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

### Hand-over-hand locking

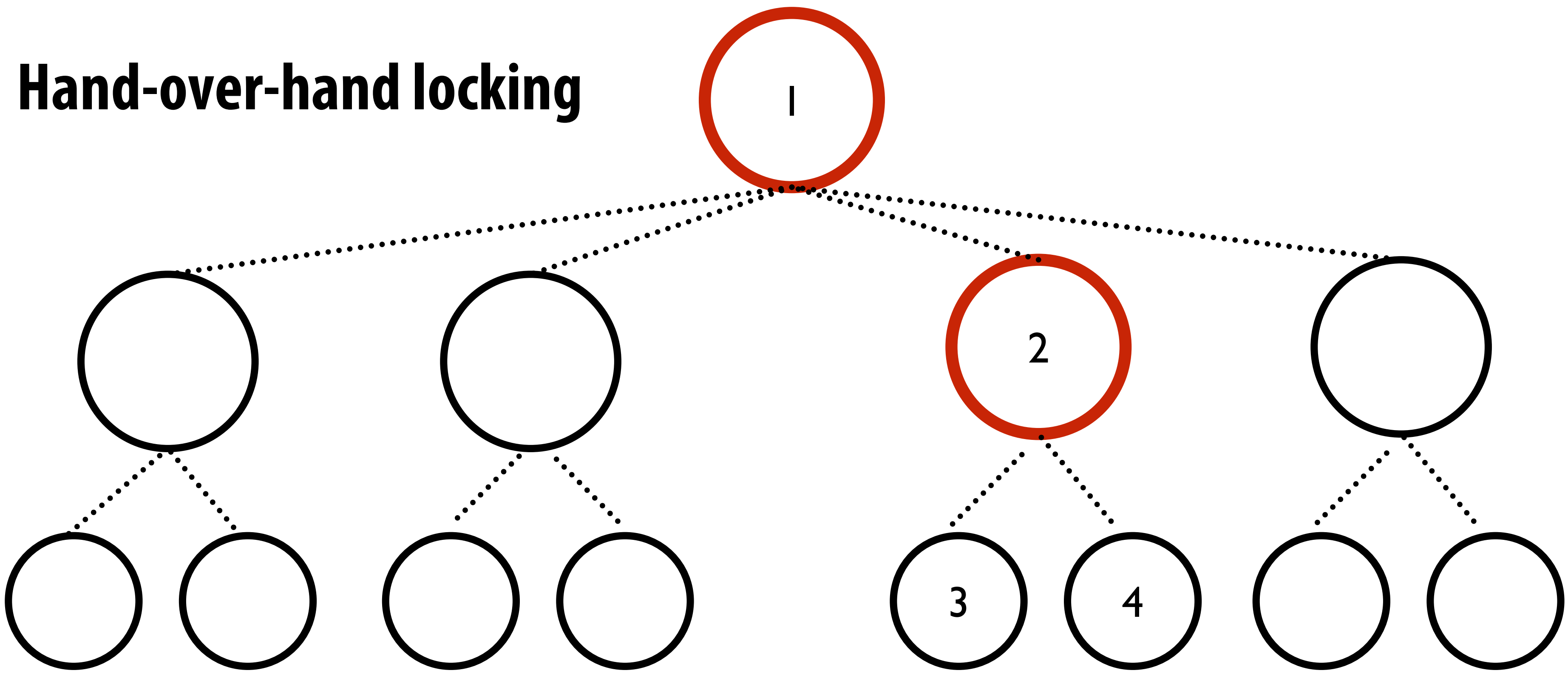



Slide credit: Austen McDonald

# Fine-grained locking example

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

### Hand-over-hand locking

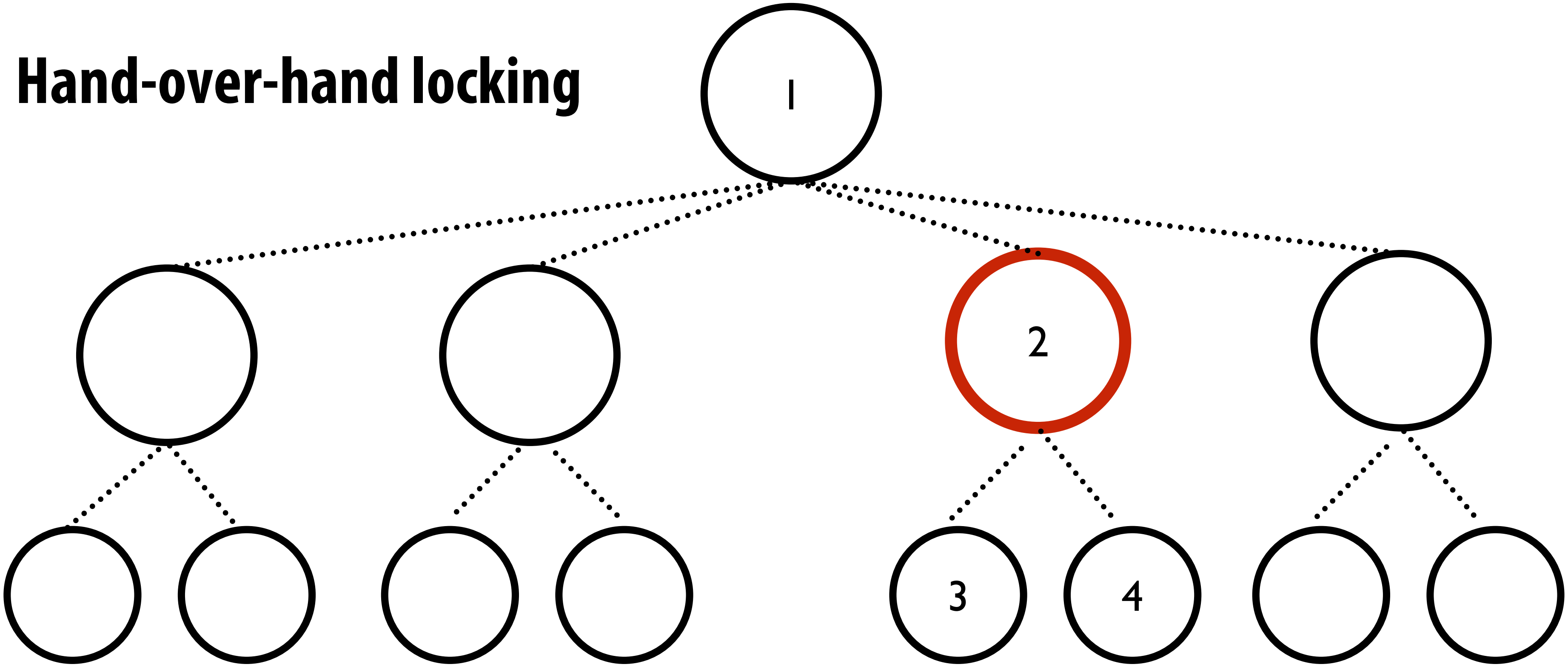



Slide credit: Austen McDonald

# Fine-grained locking example

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

### Hand-over-hand locking

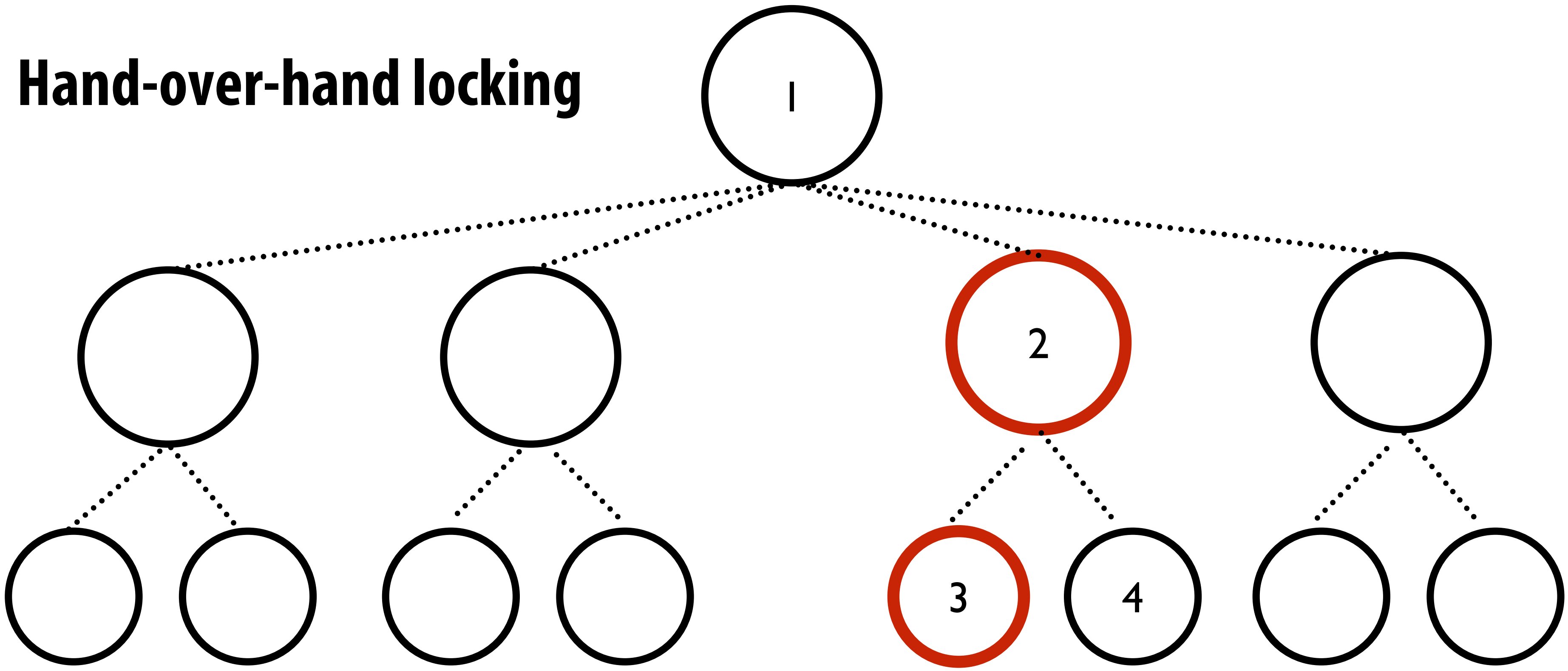



Slide credit: Austen McDonald

# Fine-grained locking example

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

### Hand-over-hand locking

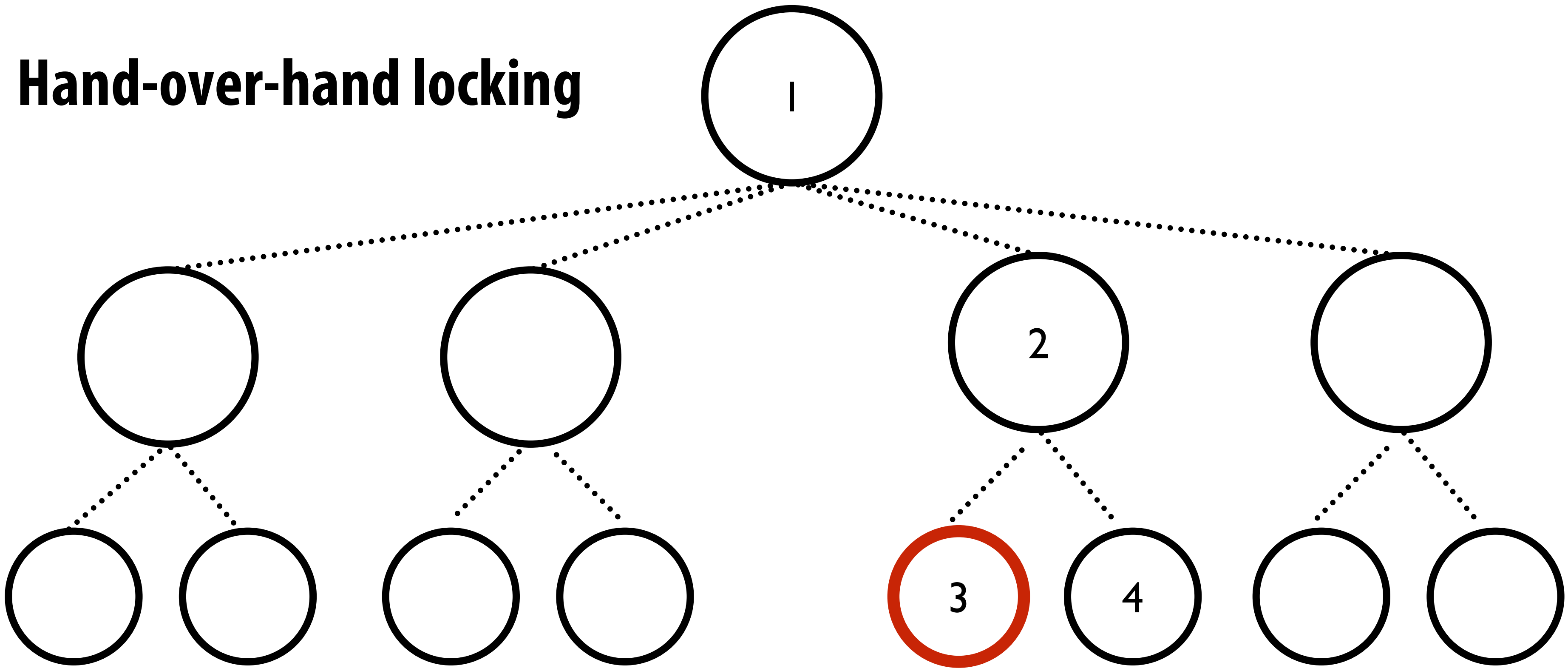



Slide credit: Austen McDonald

# Fine-grained locking example

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

### Hand-over-hand locking

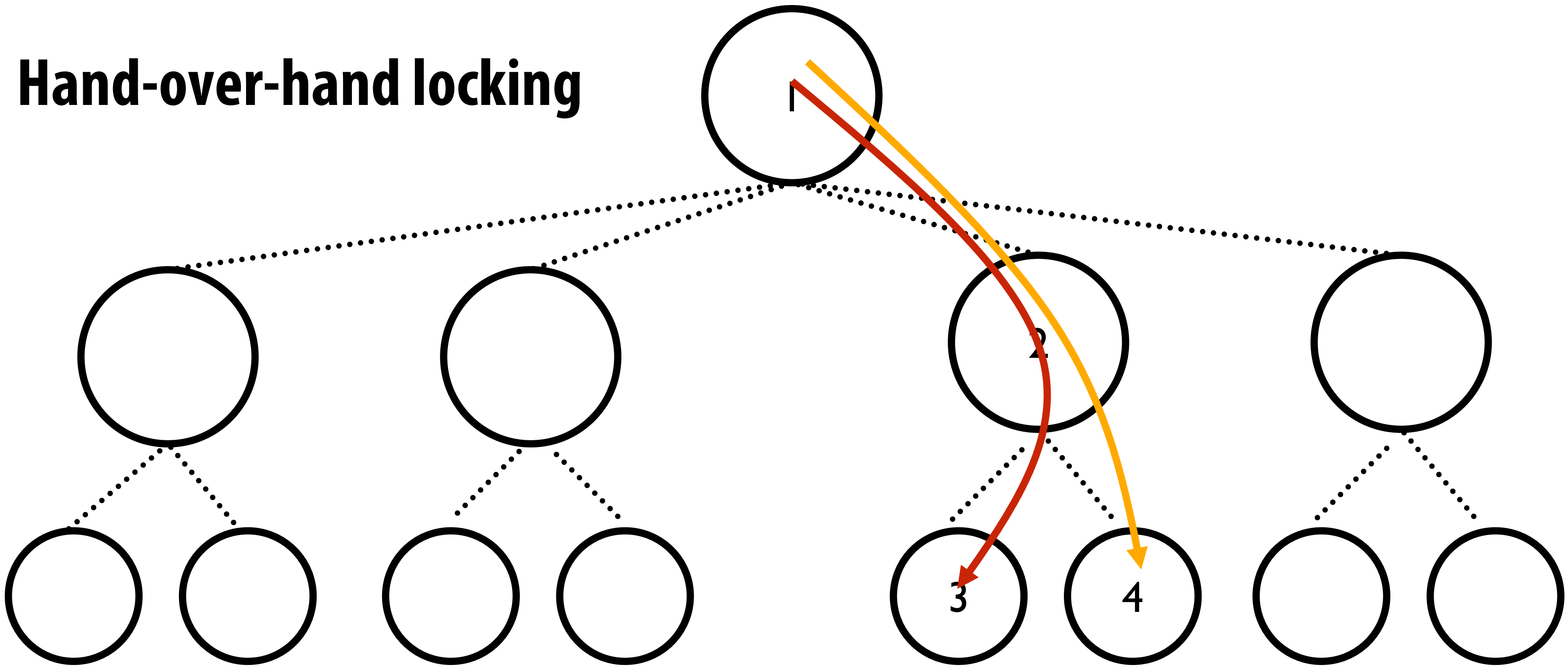



Slide credit: Austen McDonald

# Fine-grained locking example

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

### Hand-over-hand locking



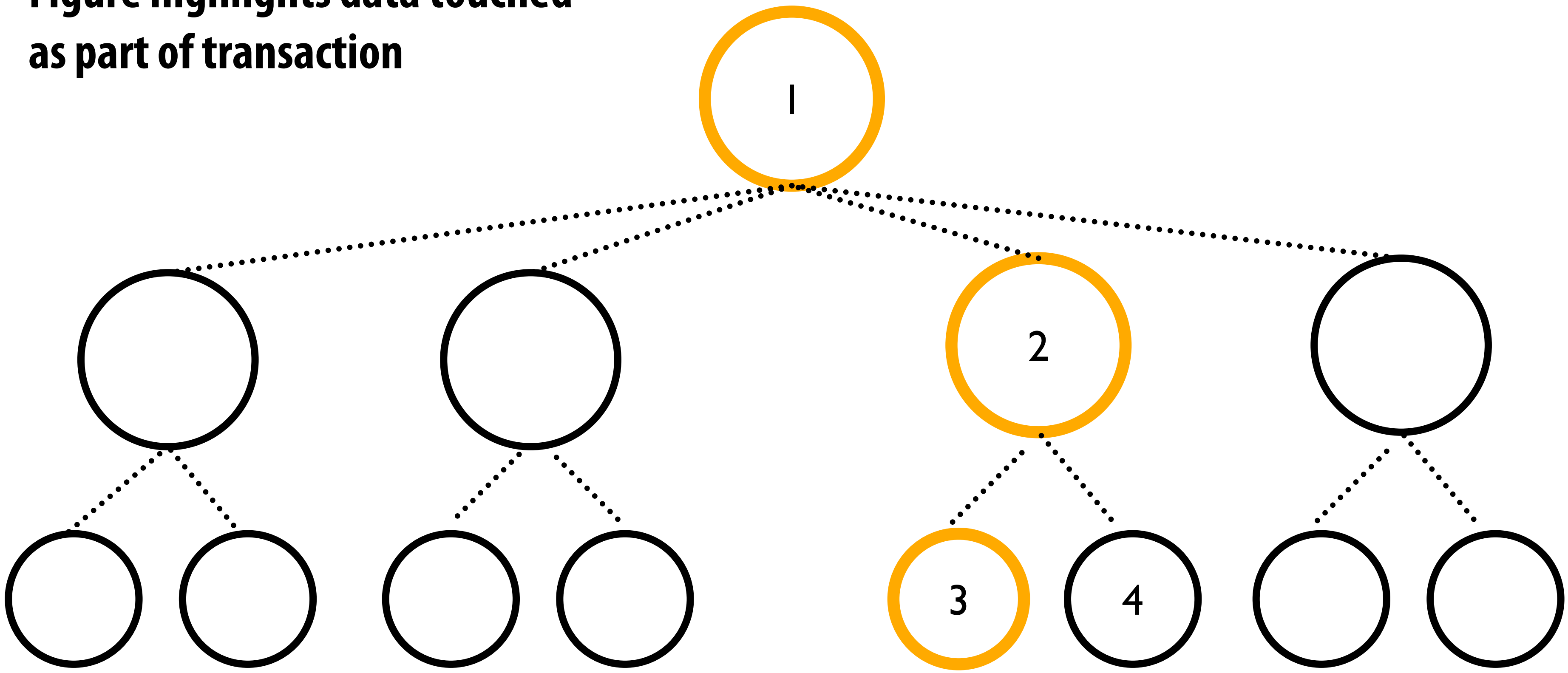

**Locking can prevent concurrency**

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

Slide credit: Austen McDonald

# Transactions example

Figure highlights data touched as part of transaction

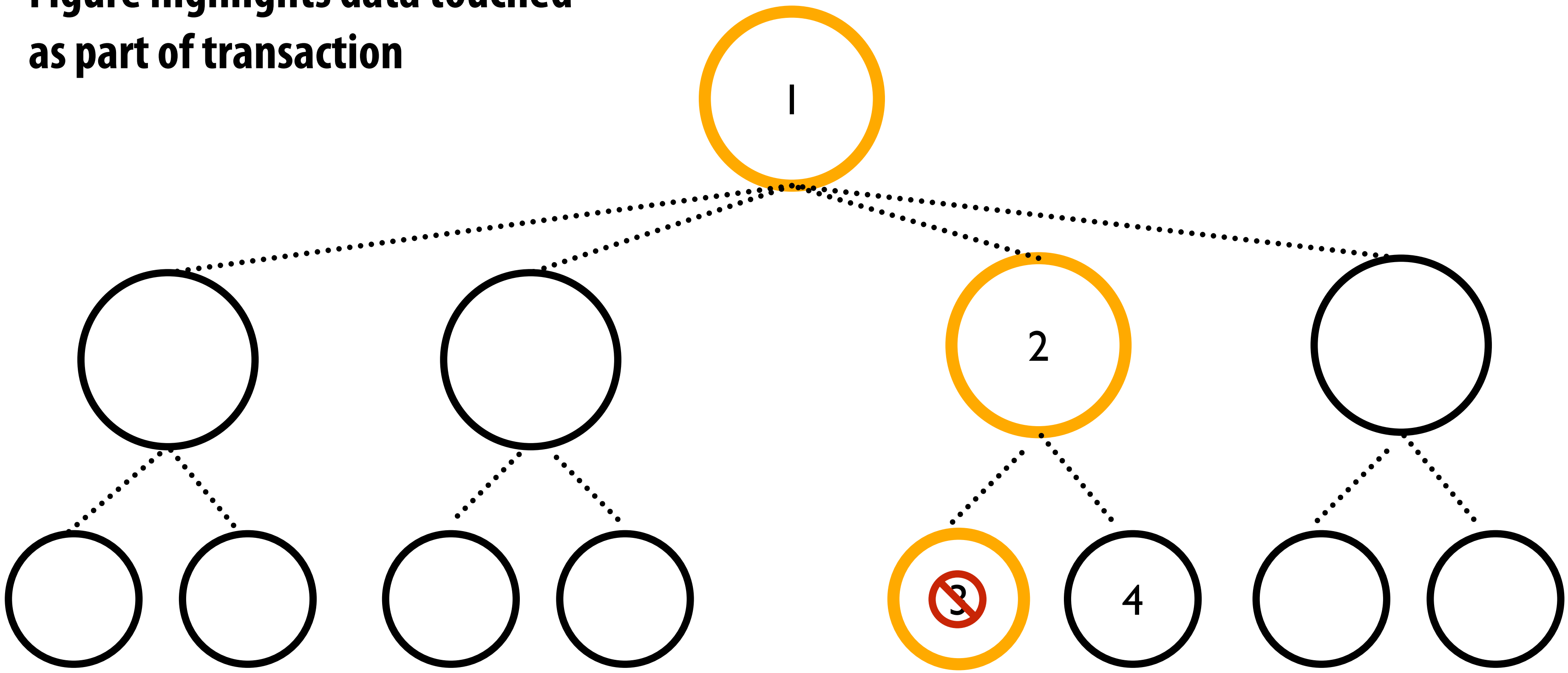



**Transaction A**
**READ: 1, 2, 3**

Slide credit: Austen McDonald

# Transactions example

Figure highlights data touched as part of transaction

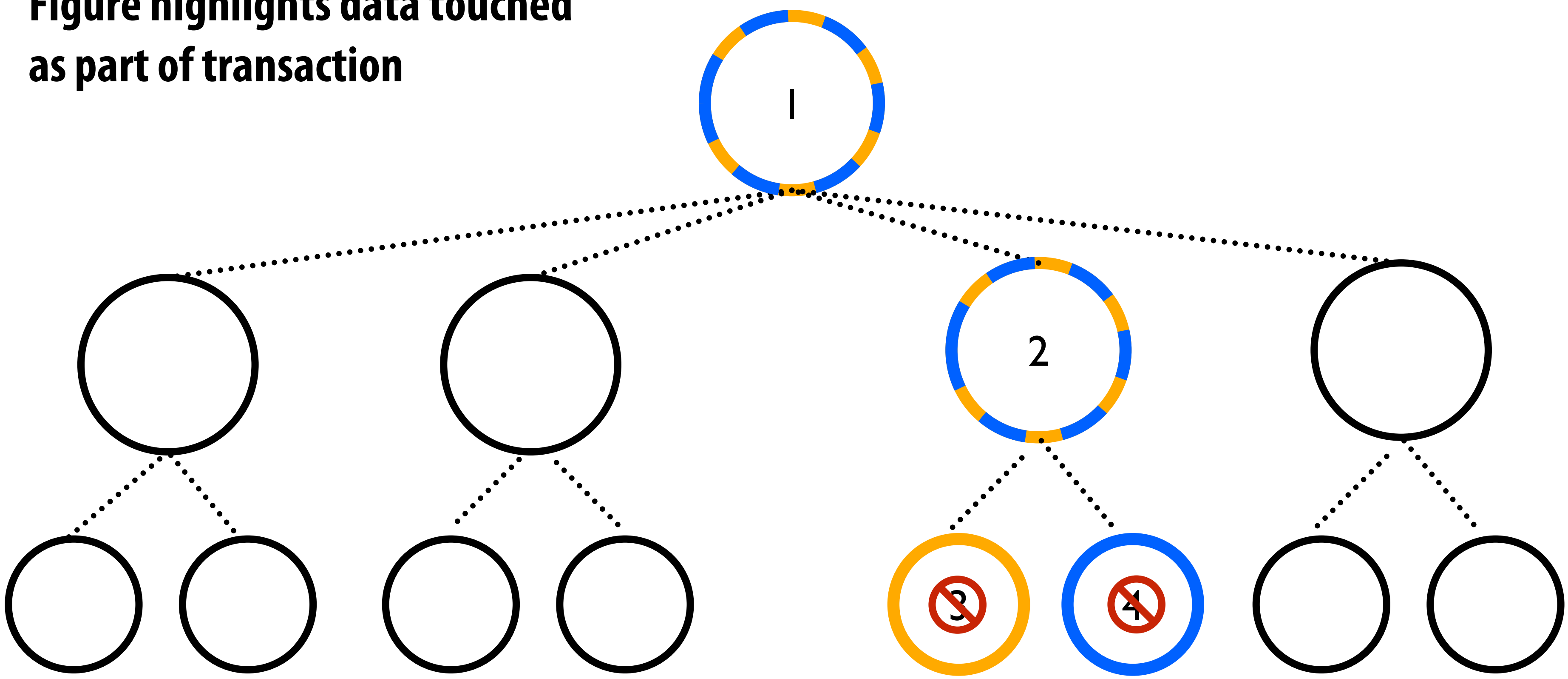



**Transaction A**
**READ: 1, 2, 3**
**WRITE: 3**

Slide credit: Austen McDonald

# Transactions example

Figure highlights data touched as part of transaction




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

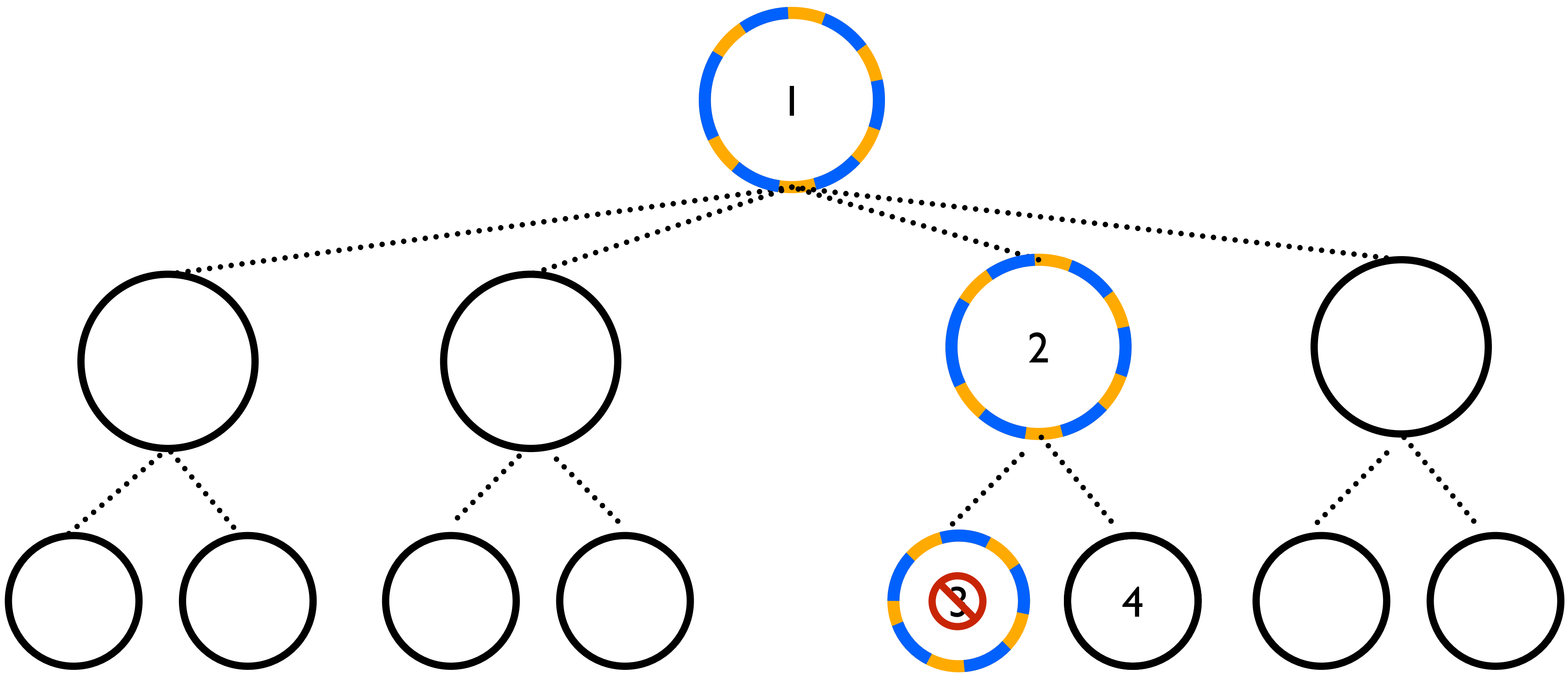
**Transaction B**
**READ: 1, 2, 4**
**WRITE: 4**

**NO READ-WRITE or WRITE-WRITE conflicts!**
(no transaction writes to data that is accessed by other transactions)

Slide credit: Austen McDonald

# Transactions example #2

(Both transactions modify node 3)




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

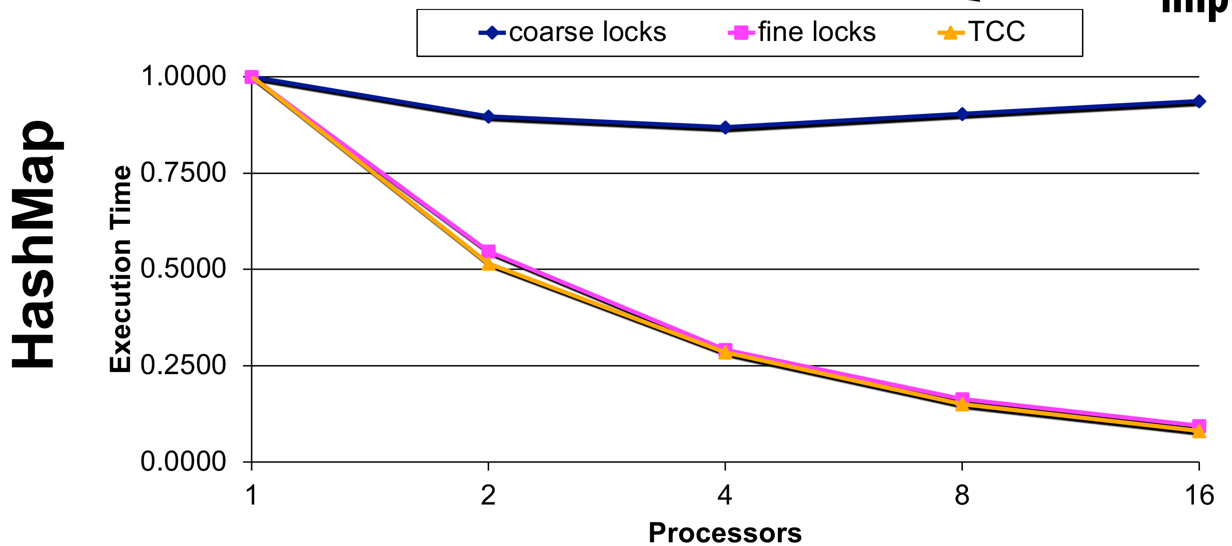
Transaction B
**READ: 1, 2, 3**
**WRITE: 3**

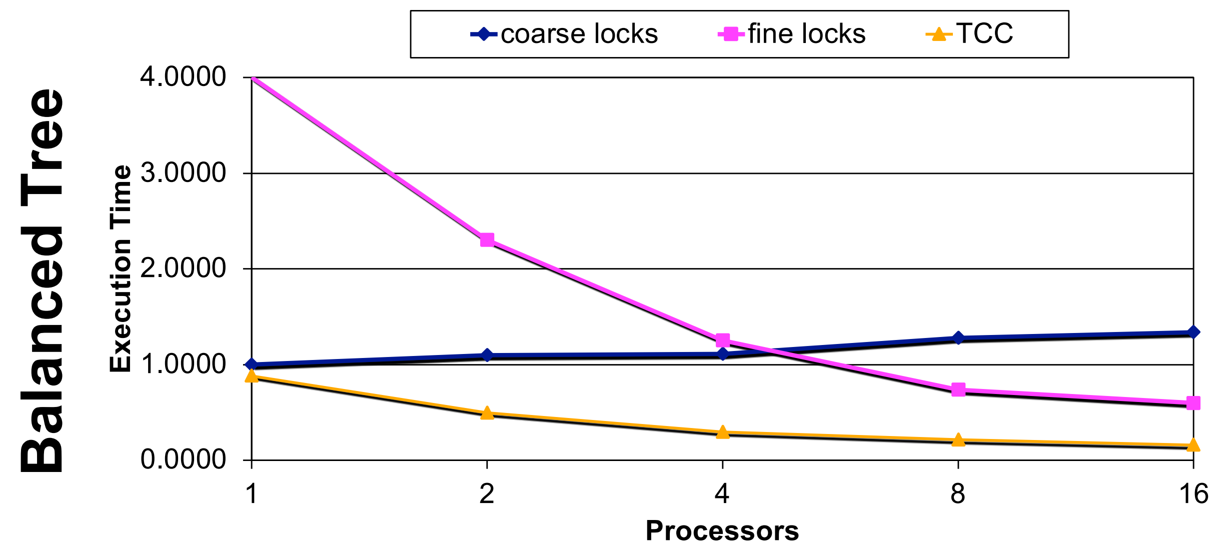
**Conflicts exist: transactions must be serialized**
**(both transactions write to node 3)**

Slide credit: Austen McDonald

# Performance: locks vs. transactions

"TCC" is a TM system implemented in hardware



# Failure atomicity: locks

```
void transfer(A, B, amount) {
   synchronized(bank)
   {
      try {
        // What if A invalid or balance too low?
        withdraw(A, amount);
        // What if B invalid?
        deposit(B, amount);
      }
      catch(withdraw_exception) { /* undo code 1*/ }
      catch(deposit_exception) { /* undo code 2*/ }
    …
   }
}
```

- Complexity of manually catching exceptions
  - Programmer provides "undo" code on a case-by-case basis
  - Complexity: must track 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 now responsible for processing exceptions
  - All exceptions (except those explicitly managed by the programmer)
  - Transaction is aborted and memory updates are undone
  - Recall: a transaction either commits or it doesn't: 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, 100);
```

**Thread 1:**

```
transfer(y, x, 100);
```

**DEADLOCK!**

- Composing lock-based code can be tricky
  - Requires system-wide policies to get correct
  - System-wide policies can break software modularity
- Programmer caught between an extra lock and a hard (to implement) place *
  - Coarse-grain locks: low performance
  - Fine-grain locking: good for performance, but can lead to deadlock

* Yes, I was particularly proud of this one.

# 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
  - System-wide policies can break software modularity
- **Programmer caught between an extra lock and a hard (to implement) place**
  - Coarse-grain locks: low performance
  - 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, 100)
```

**Thread 1:**

```
transfer(y, x, 100);
```

- **Transactions compose gracefully (in theory)**
  - Programmer declares global intent (atomic execution of transfer)
    - No need to know about global implementation strategy
  - Transaction in `transfer` subsumes any defined in `withdraw` and `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 (promise) of transactional memory

- **Easy to use synchronization construct**
  - It is difficult for programmers to get synchronization right
  - Programmer declares need for atomicity, system implements it well
  - Claim: transactions are as easy to use as coarse-grain locks
- **Often performs as well as fine-grained locks**
  - Provides automatic read-read concurrency and fine-grained concurrency
  - Performance portability: locking scheme for four CPUs may not be the best scheme for 64 CPUs
  - Productivity argument for transactional memory: system support for transactions can achieve 90% of the benefit of expert programming with fined-grained locks, with 10% of the development time
- **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 and tasks
  - TM: atomic and isolation execution
    - Easy to specify synchronization and speculation
- OpenTM features
  - Transactions, transactional loops and transactional 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 of atomicity
  - Lock: low-level blocking primitive
    - Does not provide atomicity or isolation on its own

**Make sure you understand this difference in semantics!**

- **Keep in mind**
  - Locks can be used to implement an `atomic` block but…
  - Locks can be used for purposes beyond atomicity
    - Cannot replace all uses of locks 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

# What about replacing synchronized with atomic in this example?

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

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

# Atomicity violation due to programmer error

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

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

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

- **Programmer mistake: logically atomic code sequence (in thread 1) is erroneously separated into two atomic blocks (allowing another thread to set pointer to NULL in between)**

# Implementing transactional memory

# Recall transactional semantics

- Atomicity (all or nothing)
  - At commit, all memory writes take effect at once
  - In event of 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 transaction to abort)
  - Conflict detection and resolution (WHEN to abort)
- Implementation options
  - Hardware transactional memory (HTM)
  - Software transactional memory (STM)
  - Hybrid transactional memory
    - e.g., hardware-accelerated STMs

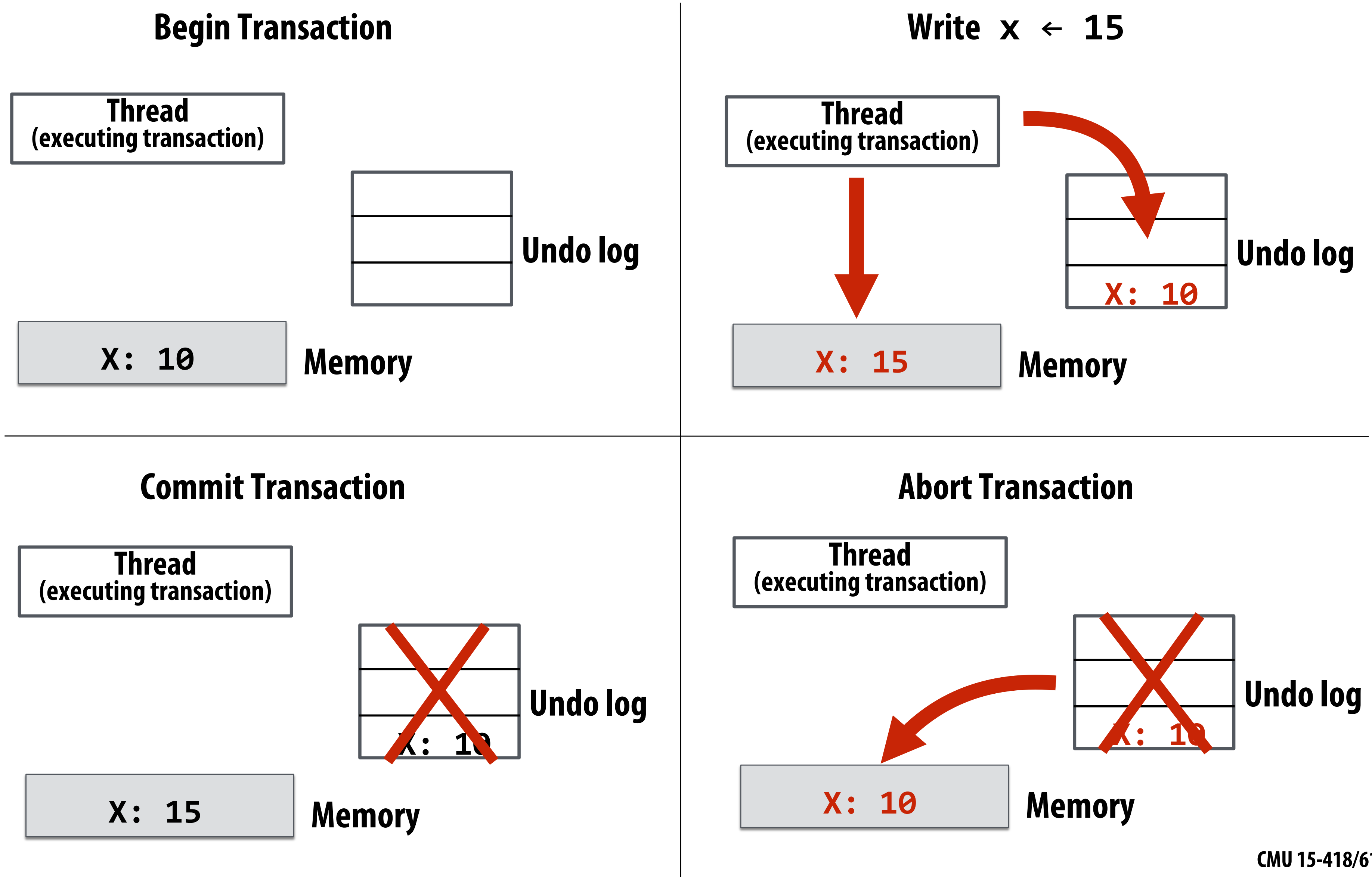
# Data versioning

**Manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions**

1. **Eager versioning (undo-log based)**
2. **Lazy versioning (write-buffer based)**

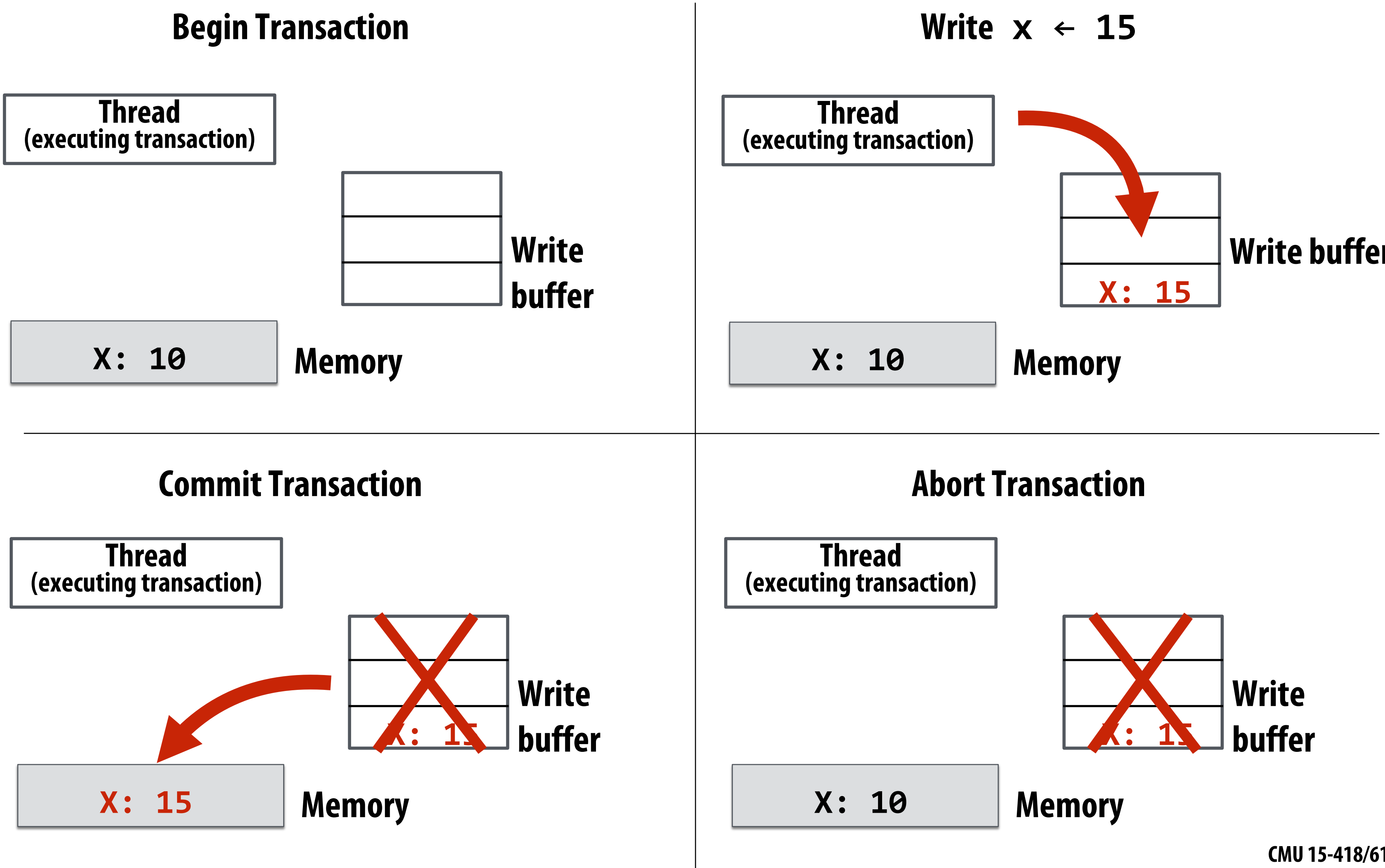
# Eager versioning (Immediate update)

## Update memory immediately, maintain “undo log” in case of abort



# Lazy versioning (Deferred update)

## 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 on write
  - Maintain undo information in a log (incurs per-store overhead)
  - Good: faster commit (data is already in memory)
  - Bad: slower aborts, fault tolerance issues (consider crash in middle of transaction)

Eager versioning philosophy: write to memory immediately, hoping transaction won't abort (but deal with aborts when you have to)

- Lazy versioning (write-buffer based)
  - Buffer data in a write buffer until commit
  - Update actual memory location on commit
  - Good: faster abort (just clear log), no fault tolerance issues
  - Bad: slower commits

Lazy versioning philosophy: only write to memory when you have to

# Conflict detection

- **Must 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 both pending, and 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 the transaction

# Pessimistic detection

- Check for conflicts during loads or stores
  - A HW implementation will check for conflicts through coherence actions

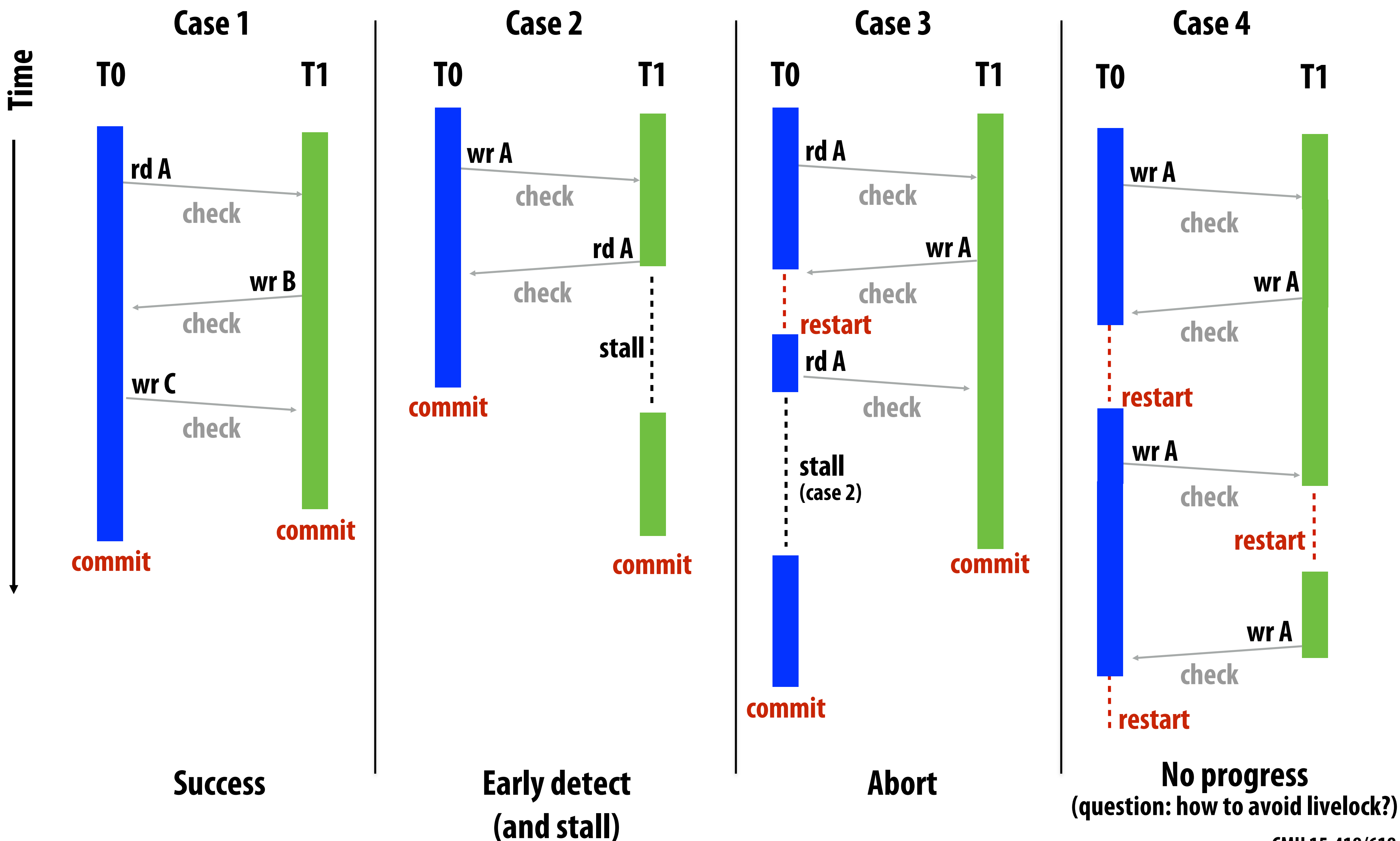
(will discuss in detail later)

  - Philosophy: “I suspect conflicts might happen, so let’s always check to see if one has occurred after each memory operation… if I’m going to have to roll back, might as well do it now to avoid wasted work.”

- “Contention manager” decides to stall or abort transaction when a conflict is detected
  - Various priority policies to handle common case fast

# Pessimistic detection examples

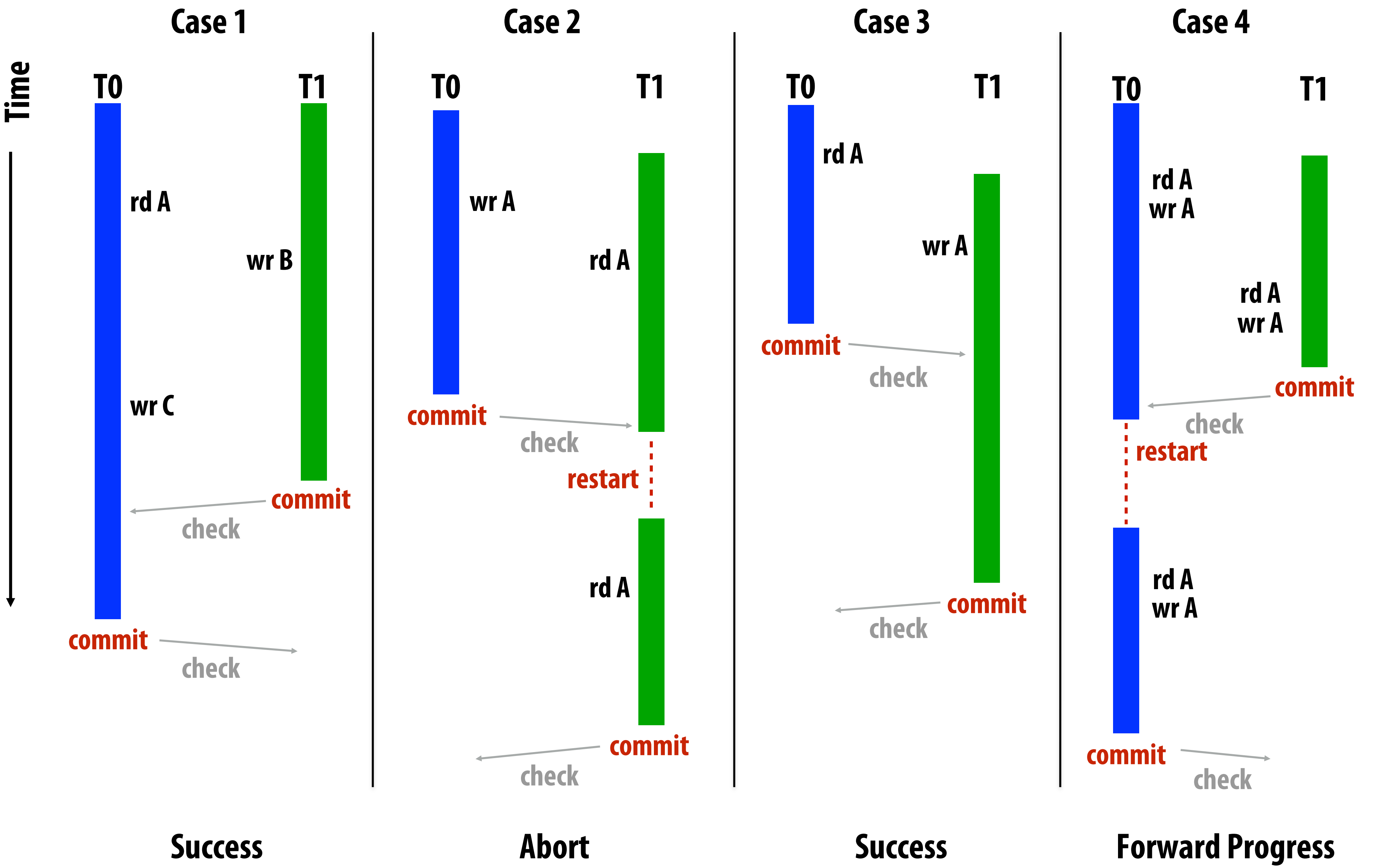
(Note: diagrams assume "aggressive" contention manager on writes: writer wins)



# 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
  - Intuition: "Let's hope for the best and sort out all the conflicts only when the transaction tries to commit"
- **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. "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 (check on each load/store)
  - Bad: detection on critical path

- **Optimistic conflict detection (a.k.a. "lazy" or "commit")**
  - Good: forward progress guarantees
  - Good: bulk communication and conflict detection
  - 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
  - Bad: 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 and word
- Can mix and match to get best of both worlds
  - Word-level for arrays, object-level for other data, …

# TM implementation space (examples)

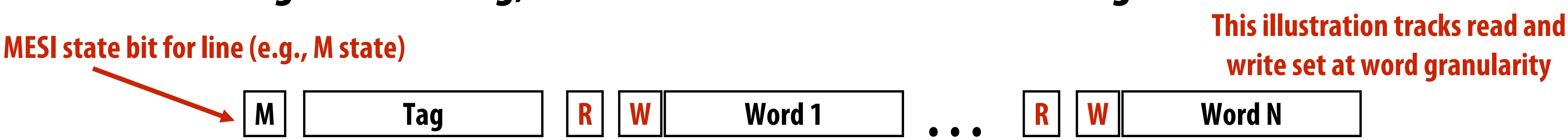
- **Hardware TM systems**
  - Lazy + optimistic: Stanford TCC, Intel VTM
  - Lazy + pessimistic: MIT LTM
  - Eager + pessimistic: Wisconsin LogTM
  - Eager + optimistic: not practical
- **Software TM systems**
  - Lazy + optimistic (rd/wr): Sun TL2
  - Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
  - 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 is implemented in caches
  - Cache the write buffer or the undo log
  - Add new cache line metadata to track transaction read set and write set
- Conflict detection through cache coherence protocol
  - Coherence lookups detect conflicts between transactions
  - Works with snooping and directory coherence
- Note:
  - Register checkpoint must also be taken at transaction begin (to restore execution context state on abort)

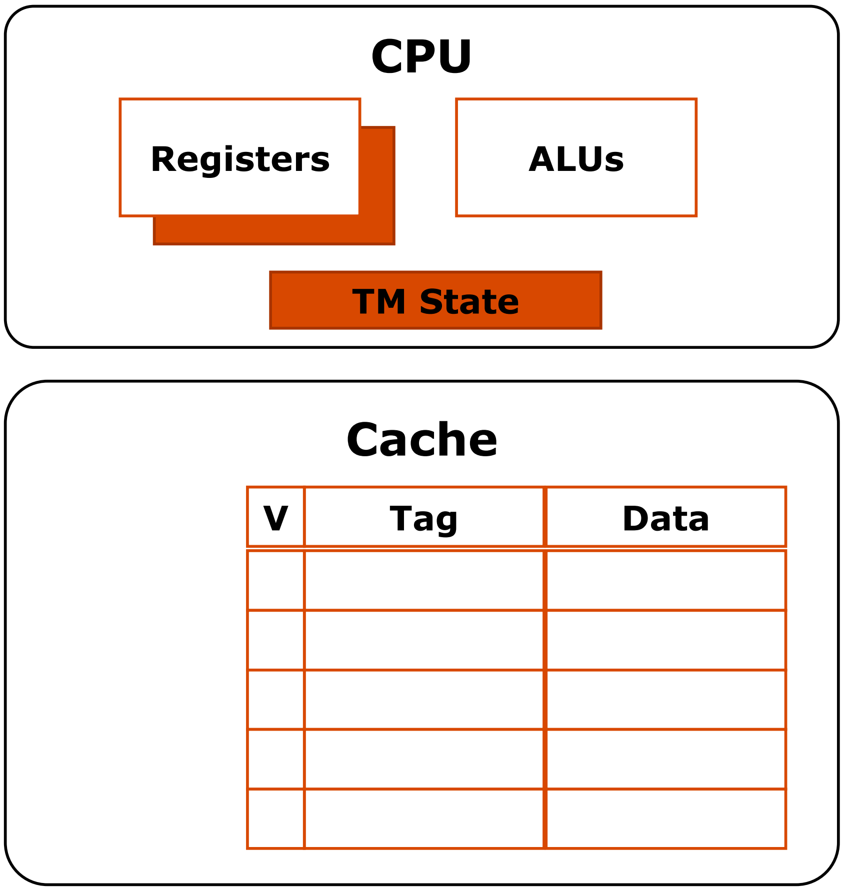
# HTM design

- **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



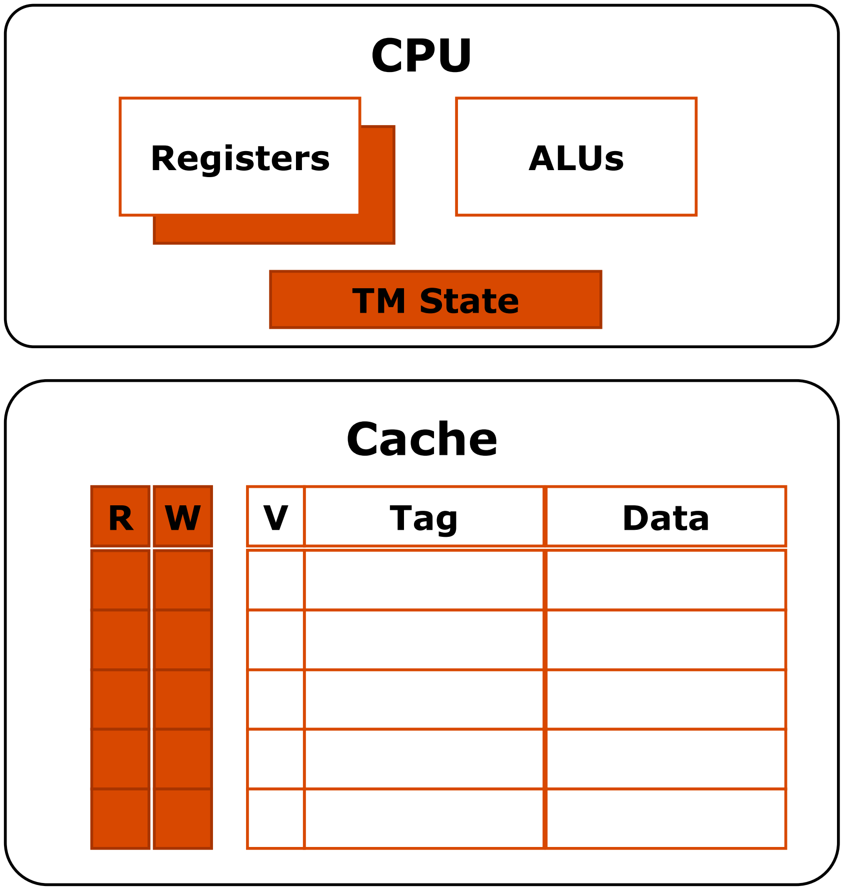

- **Coherence requests check R/W bits to detect conflicts**
  - Observing shared request to W-word is a read-write conflict
  - Observing exclusive (intent to write) request to R-word is a write-read conflict
  - Observing exclusive (intent to write) request to W-word is a write-write conflict

# Example HTM implementation: lazy-optimistic



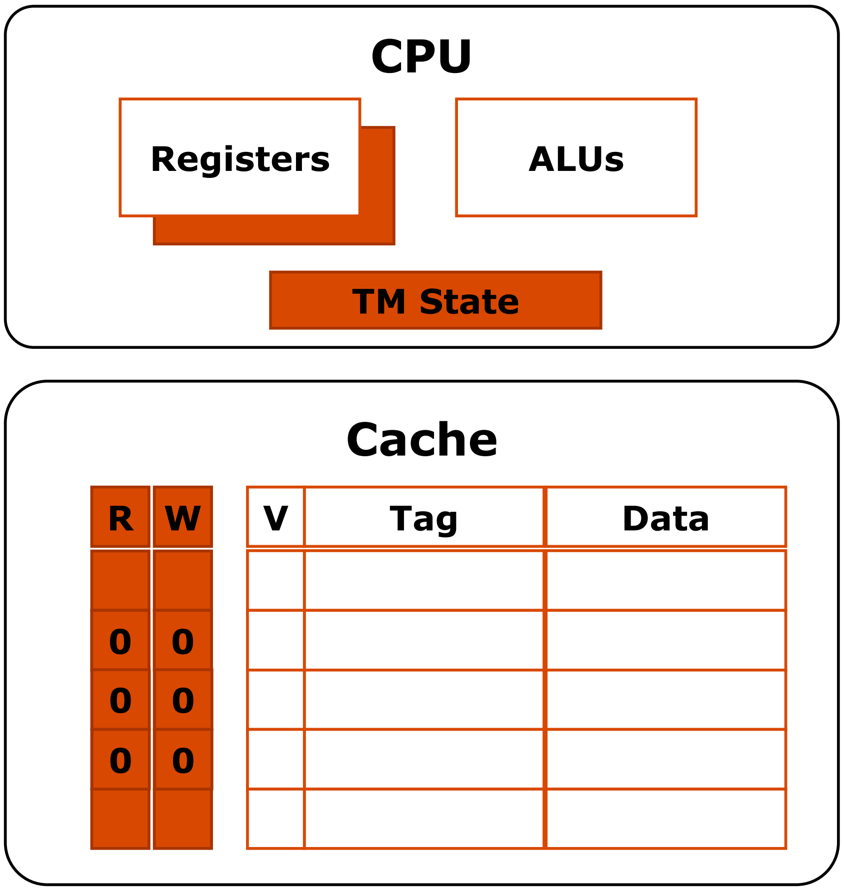

- **CPU changes**
  - Ability to checkpoint register state (available in many CPUs)
  - TM state registers (status, pointers to abort handlers, ...)

# Example HTM implementation: lazy-optimistic




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

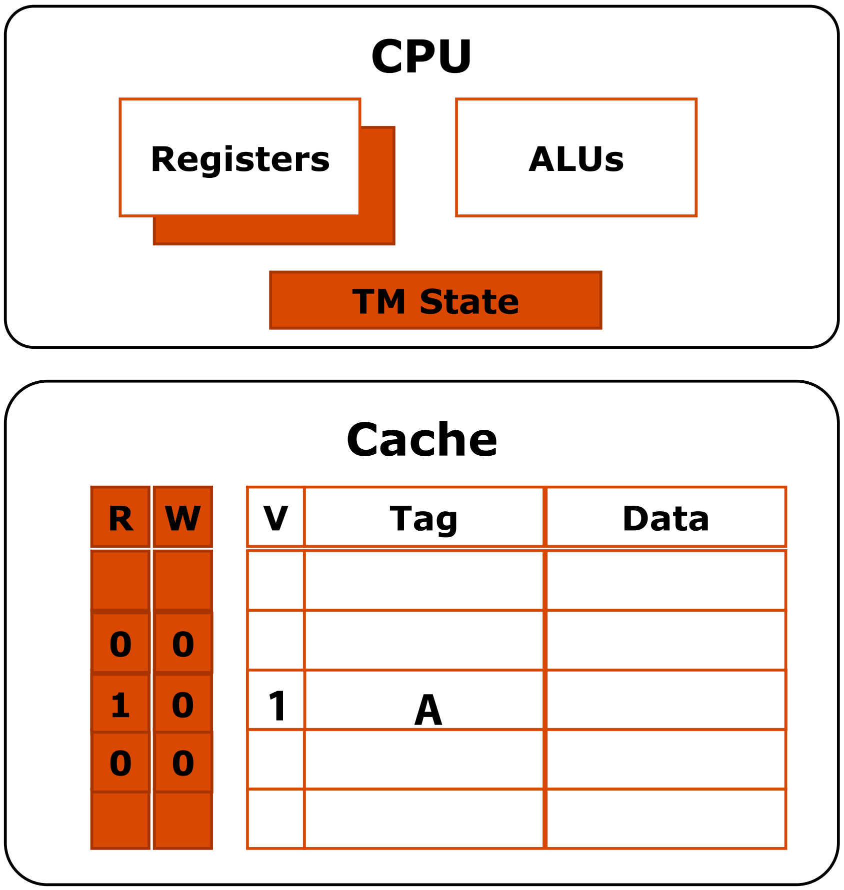
# HTM transaction execution




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

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

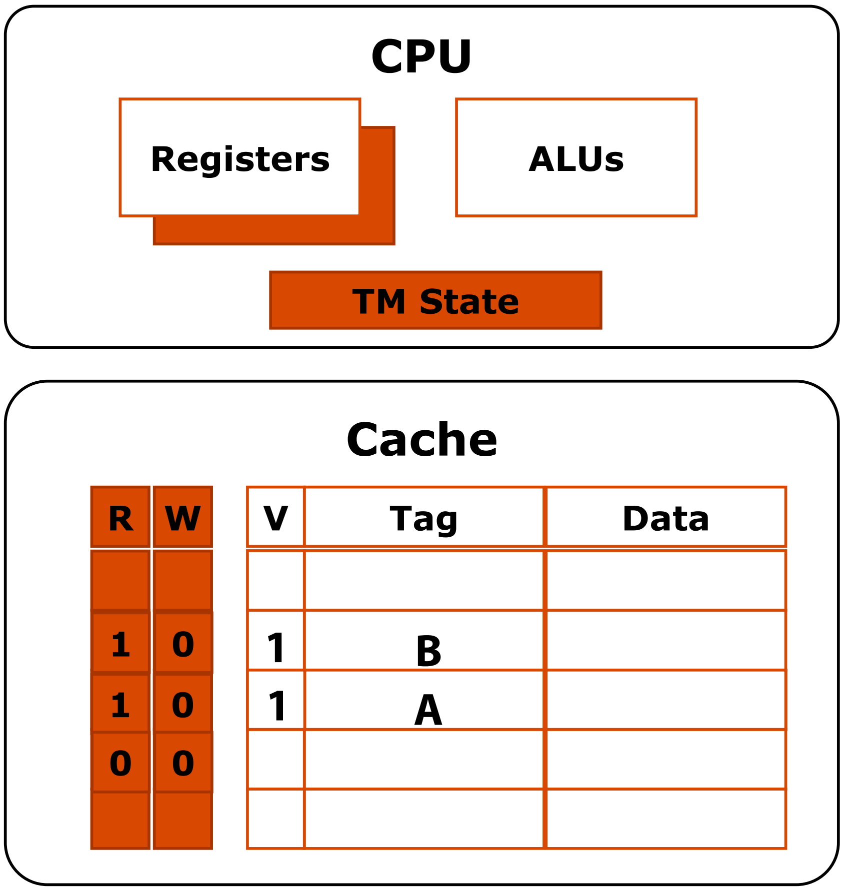
# HTM transaction execution




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

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

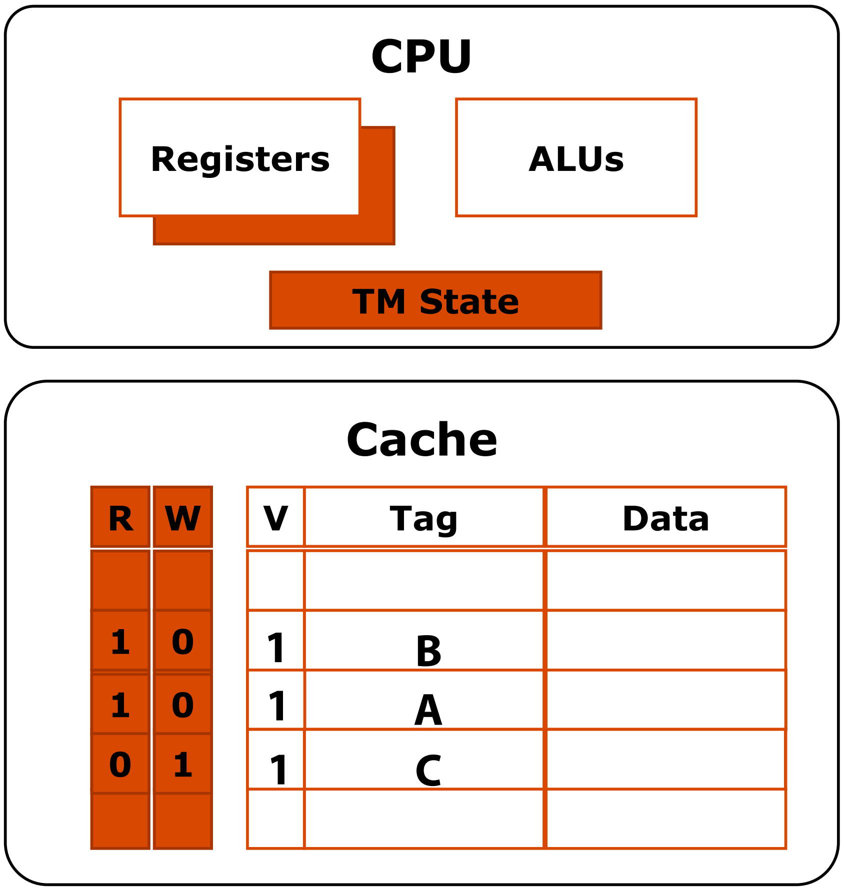
# HTM transaction execution




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

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

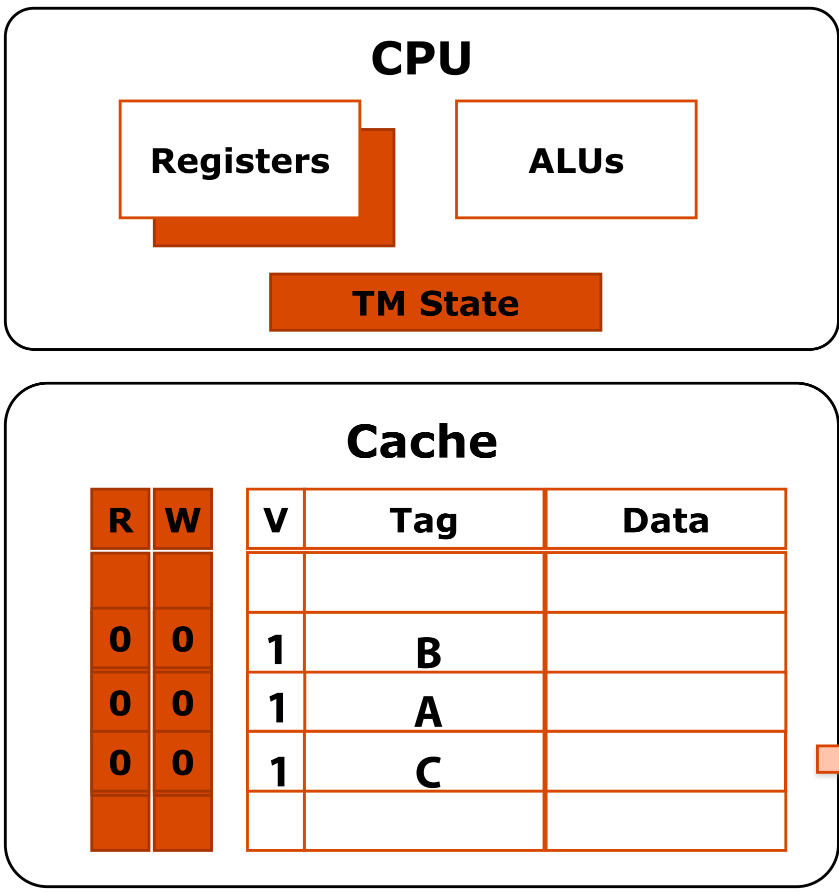
# HTM transaction execution




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

- **Store operation**
  - Service cache miss if needed
  - Mark data as part of write set (note: this is not a load into exclusive state. Why?)

# HTM transaction execution: commit



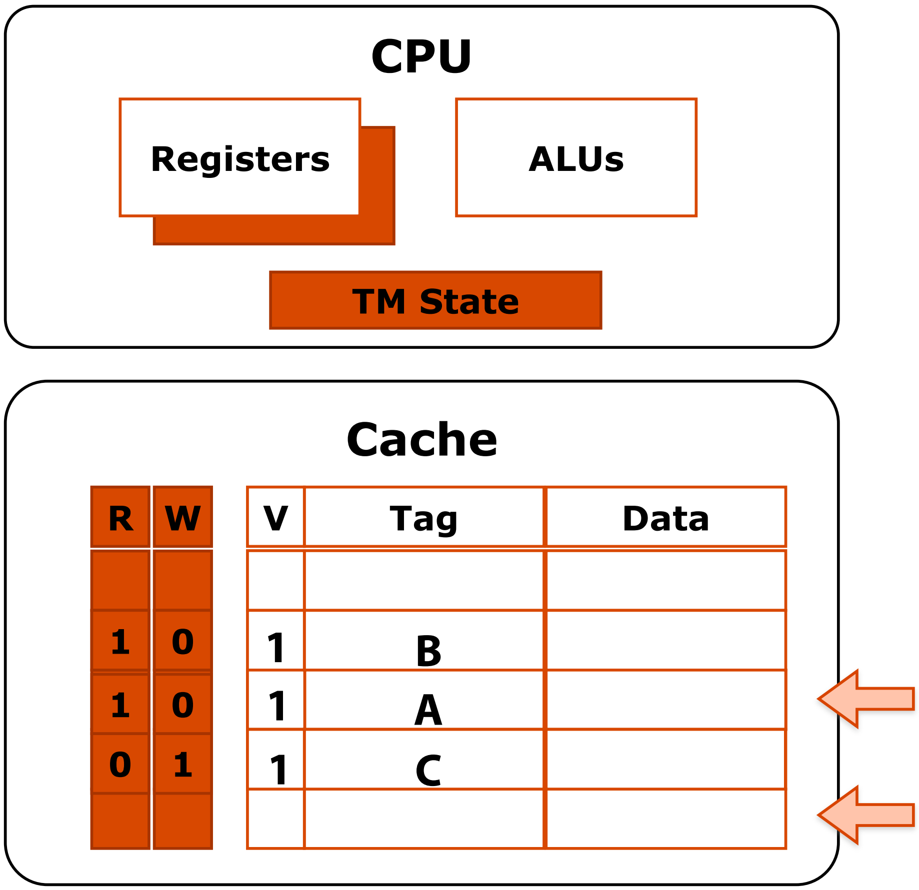

```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit  ⬅
```

upgradeX C
(result: C is now in exclusive-dirty state)

- **Fast two-phase commit**
  - Validate: request RdX 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

## Assume remote processor commits transaction with writes to A and D




```
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 to register checkpoint

# Hardware transactional memory support in Intel Haswell architecture *

- **New instructions for "restricted transactional memory" (RTM)**
  - `xbegin`: takes pointer to "fallback address" in case of abort
    - e.g., fallback to code-path with a spin-lock
  - `xend`
  - `Xabort`
  - Implementation: tracks read and write set in L1 cache
- **Processor makes sure all memory operations commit atomically**
  - But processor may automatically abort transaction for many reasons (e.g., eviction of line in read or write set will cause a transaction abort).
    - Implementation does not guarantee progress (see fallback address)
  - Intel optimization guide (ch 12) gives guidelines for increasing probability that transactions will not abort

* Shipped with bug that caused Intel disable it when discovered in 2014, fixed in Broadwell arch chips

# GCC Support

- **_xbegin(), _xend(), _xabort() + macros**

```
#include <immintrin.h>
int n_tries, max_tries;
unsigned status = _XABORT_EXPLICIT; ...

for (n_tries = 0; n_tries < max_tries; n_tries++)  {
    status = _xbegin ();
    if (status == _XBEGIN_STARTED || !(status & _XABORT_RETRY))
        break;
}

if (status == _XBEGIN_STARTED) {
    ... transaction code...
    _xend ();
} else {
     ... non-transactional fallback path...
}
```

# TSX does not guarantee progress

- Transactions fail for many reasons
- Writing fallback paths still require locks
  - The fallback path most overlap with the transaction
  - The lock path must prevent transactions from committing

- For example:

```
Result status = _xbegin();
if (status == SUCCESS) {
    if (_stop_the_world) {
        _xabort();
    }
    ...
    _xend();
}
```

```
else {
    /* Fall back path */
    lock();
    _stop_the_world = true;
    ...
    _stop_the_world = false;
    unlock();
}
```

Results collected by Mario Dehesa-Azuara and Nick Stanley as Spring 2016 project

# TSX Performance

- TSX can only track a limited number of locations
  - Minimize memory touched
- Transactions have a cost
  - Approximately equal to the cost of six atomic primitives to the same cache line

Results collected by Mario Dehesa-Azuara and Nick Stanley as Spring 2016 project

# Summary: transactional memory

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