计算机系统分析与性能评价

Computer Systems Analysis and Performance Evaluation

陈进才 周健 李国宽

武汉光电国家研究中心

光电信息存储研究部

Transactional Memory

Overview

Introduction

Advantages of TM

Implementing TM

Abstraction for synchronization

- Raising level of abstraction for synchronization
- Machine-level synchronization prims:
  - fetch-and-op, test-and-set, compare-and-swap
- We used these primitives to construct higher level, but still quite basic, synchronization prims:
  - lock, unlock, barrier
- Today:
  - transactional memory: higher level synchronization

What you should know

- What a transaction is
- The difference between the atomic construct and locks
- Design space of transactional memory implementations
  - data versioning policy
  - con" ict detection policy
  - granularity of detection
- Understand HW implementation of transaction memory

Lock/Unlock Example

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

- 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 ensure atomicity (ensures mutual exclusion on the account)

Programming with TM

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

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

- Declarative synchronization
  - Programmers says what but not how
  - No explicit declaration or management of locks
- System implements synchronization
  - Typically with optimistic concurrency
  - Slow down only on true conflicts (R-W or W-W)

Declarative vs. Imperative Abs.

- Declarative: programmer de# nes what should be done
  - Process all these 1000 tasks
  - Perform this set of operations atomically

- Imperative: programmer states how it should be done
  - Spawn N worker threads. Pull work 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)
  - 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

Advantages of TM

Example: Java 1.4 HashMap

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

- Map: Key → Value
  - Not thread safe
  - But no lock overhead when not needed

Synchronized HashMap

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

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

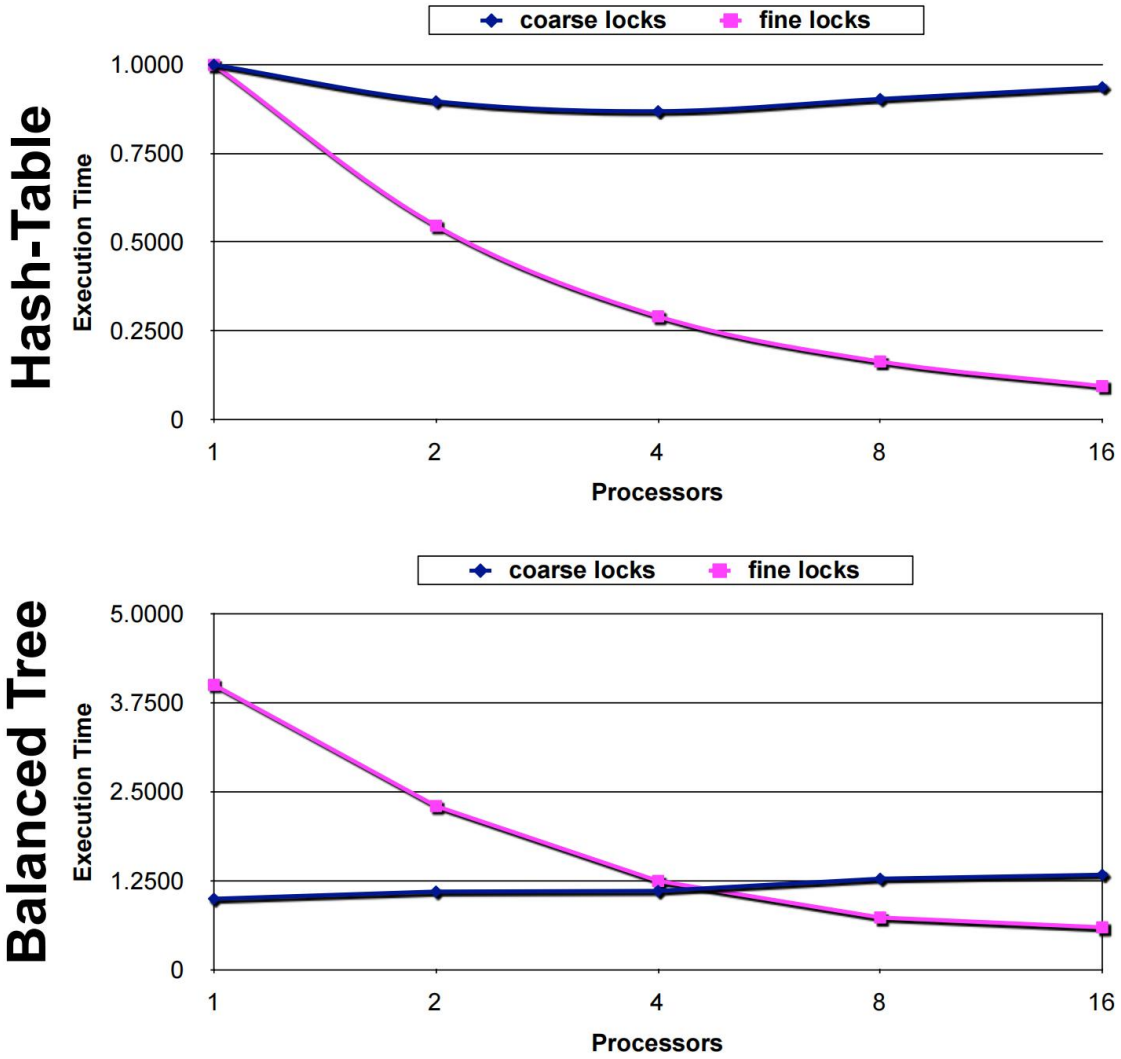
- Coarse-grain synchronized HashMap
  - Pros: thread-safe, easy to program
  - Cons: limits concurrency, poor scalability
- Only one thread can operate on map at any time

Better solution?

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

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

Performance: Locks



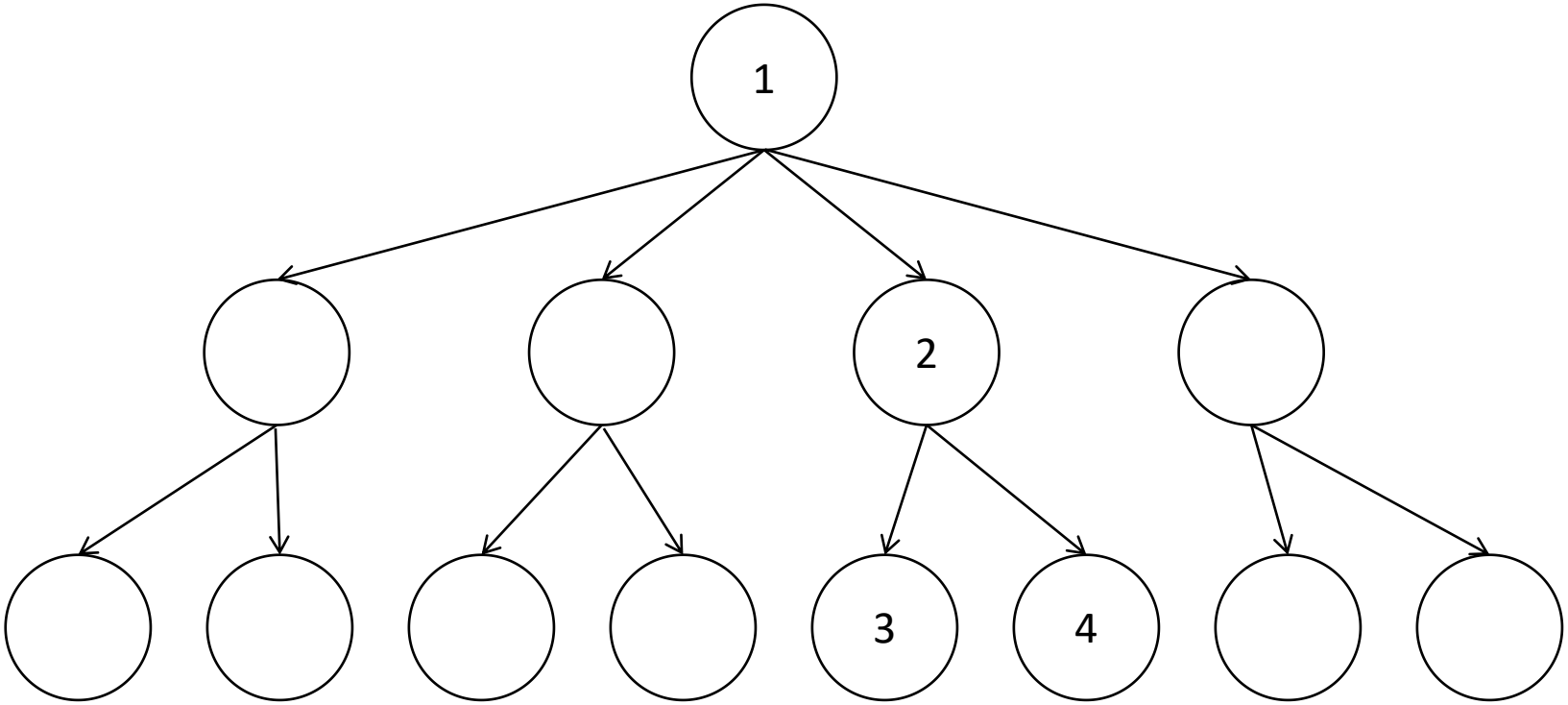
Transactional HashMap

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

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

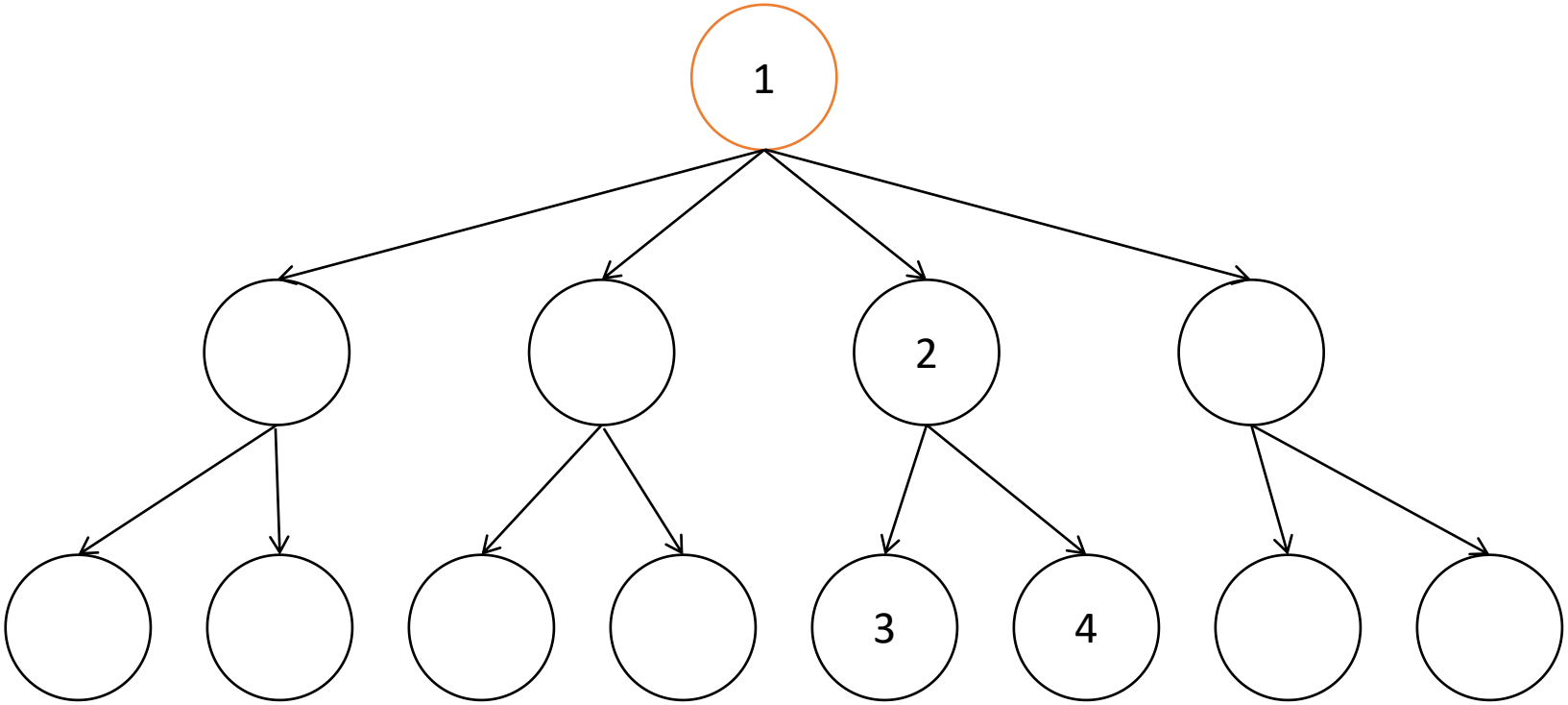
- Transactional HashMap
  - Pros: thread-safe, easy to program
  - Q: good performance & scalability?
- Depends on the implementation, but typically yes

Synchronization Example



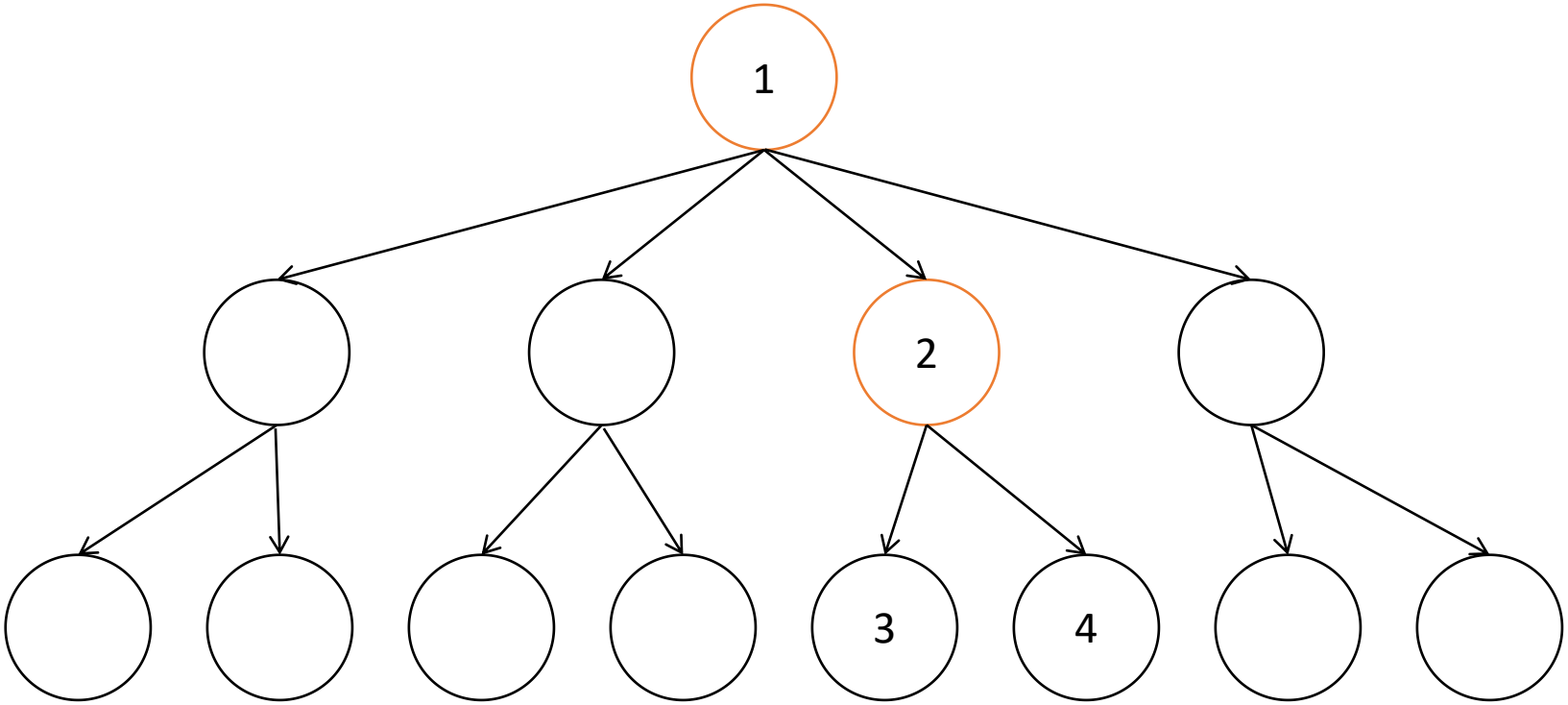
- Goal: Modify node 3 in a thread-safe way.

Synchronization Example



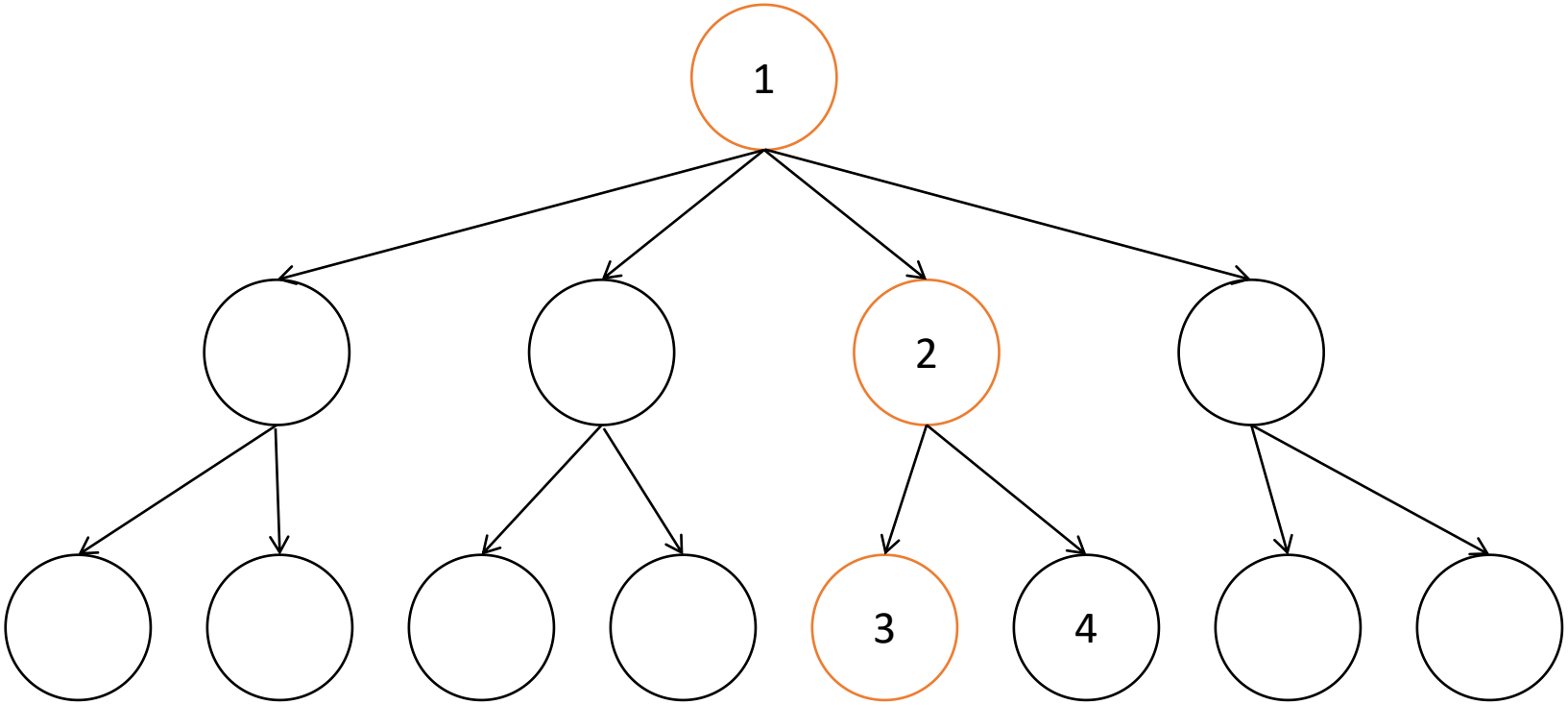
- Goal: Modify node 3 in a thread-safe way.

Synchronization Example



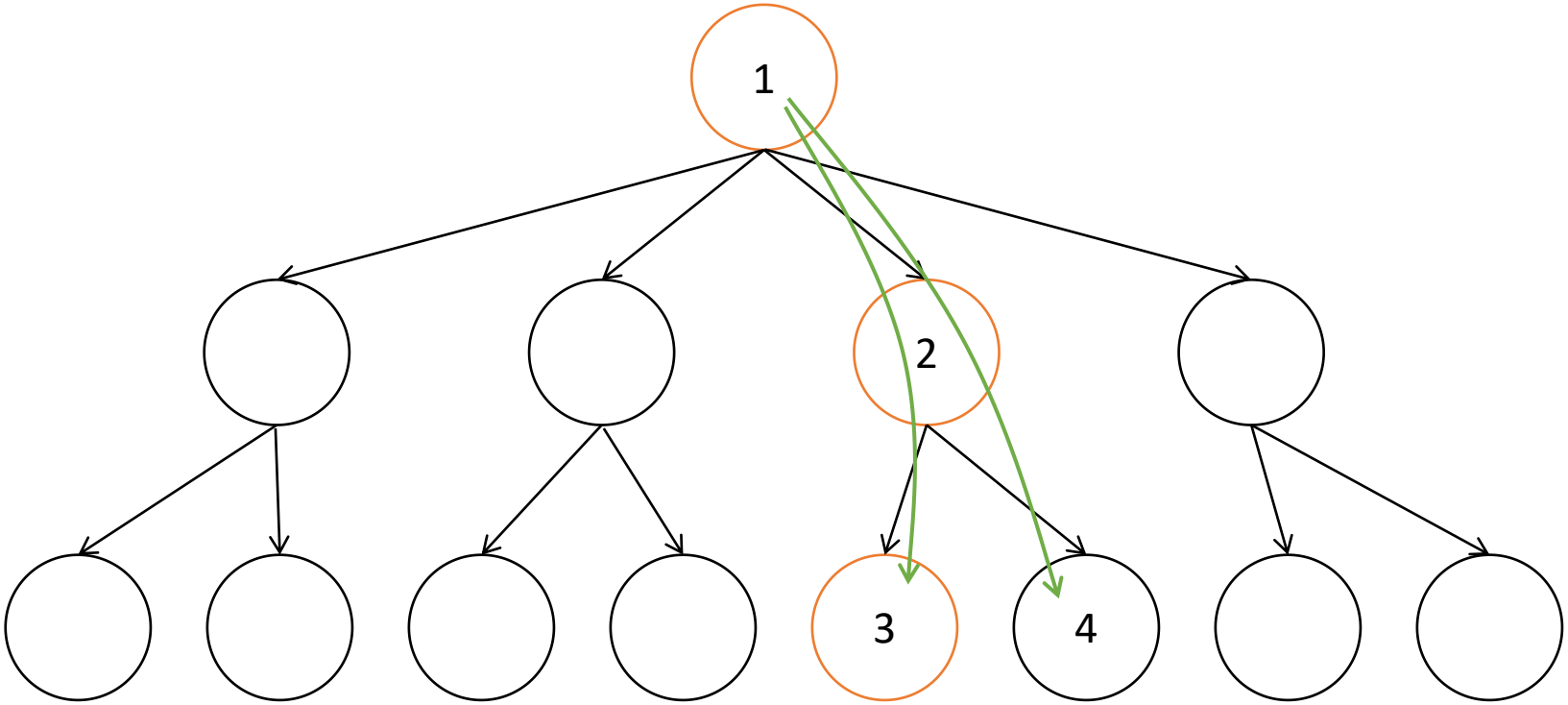
- Goal: Modify node 3 in a thread-safe way.

Synchronization Example



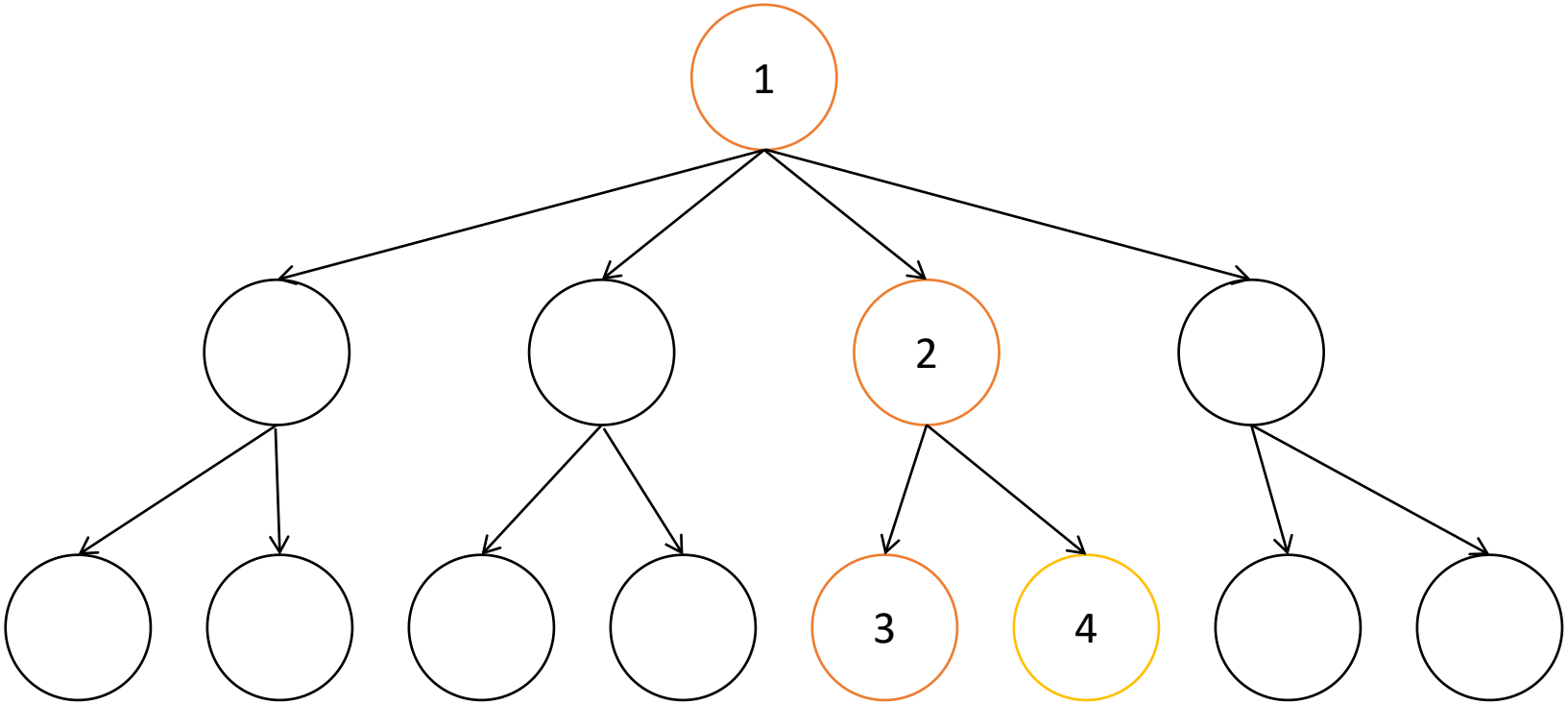
- Goal: Modify node 3 in a thread-safe way.

Synchronization Example



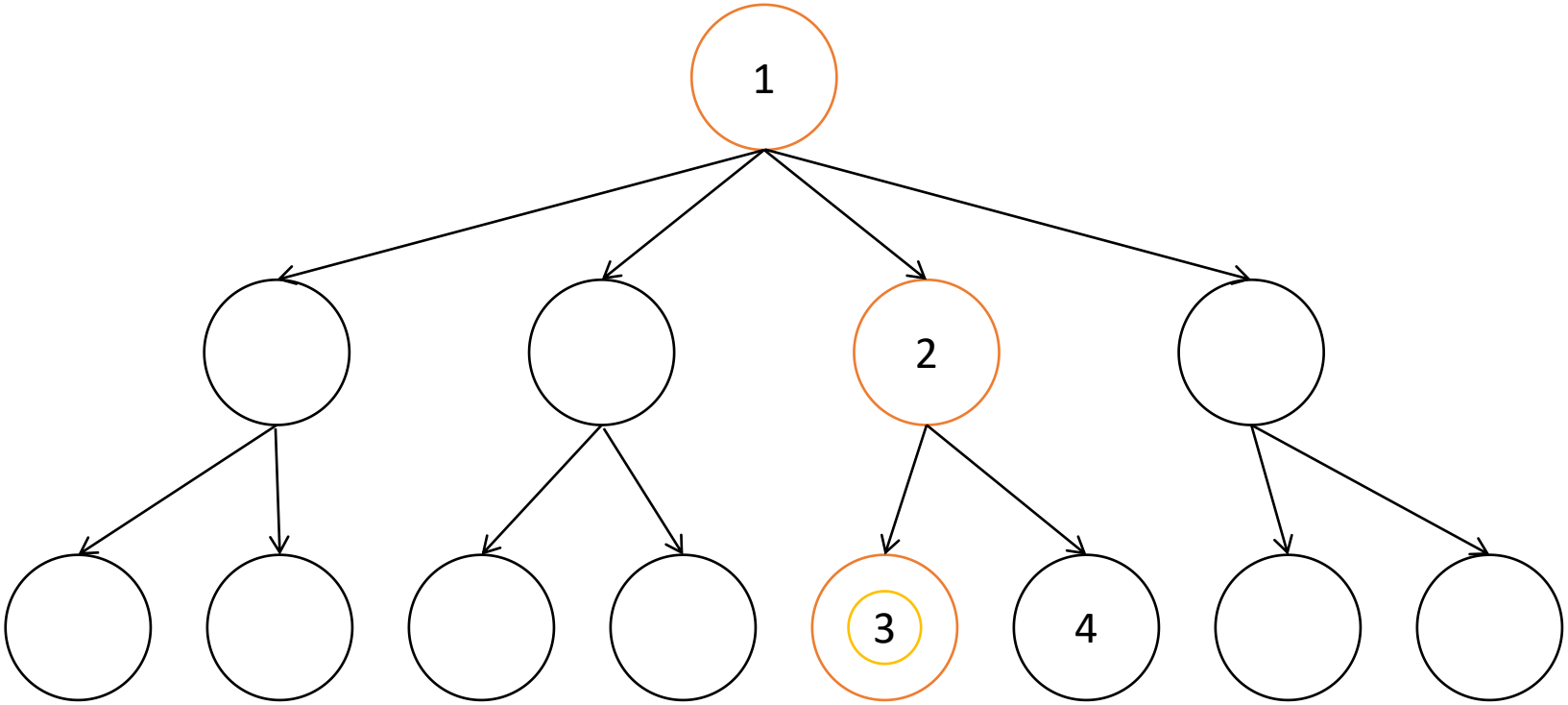
- Goal: Modify node 3 and 4 in a thread-safe way.
- Locking prevents concurrency

TM Example: No Conflicts



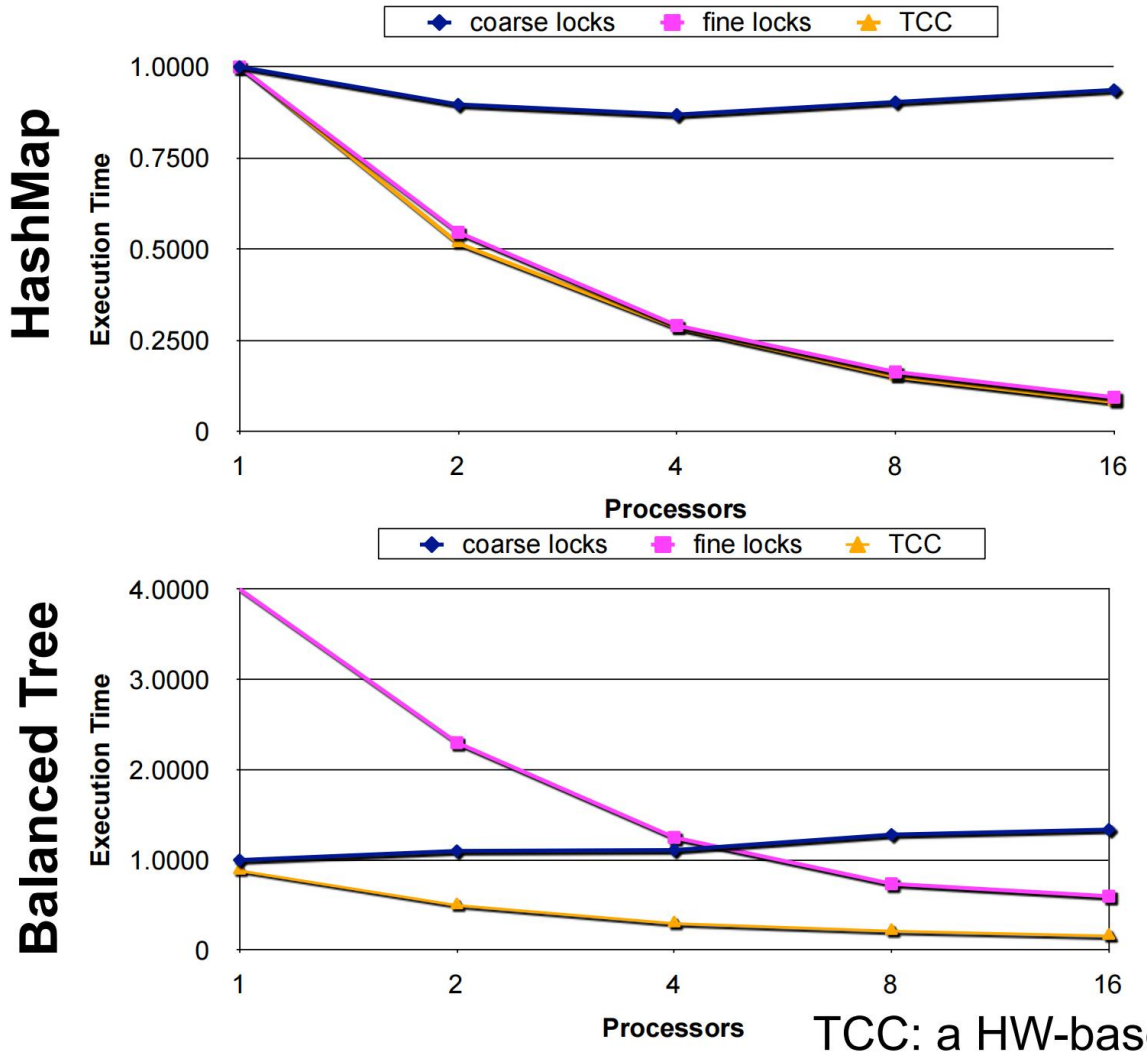
- Goal: Modify node 3 and 4 in a thread-safe way.
  - Transaction A: Read 1 2 3; Write 3
  - Transaction B: Read 1 2 4; Write 4

TM Example: With Conflicts



- Goal: Modify node 3 and 4 in a thread-safe way.
  - Transaction A: Read 1 2 3; Write 3
  - Transaction B: Read 1 2 3; Write 3

Perf.: locks vs. transactions



TCC: 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*/ }
    …
   }
```

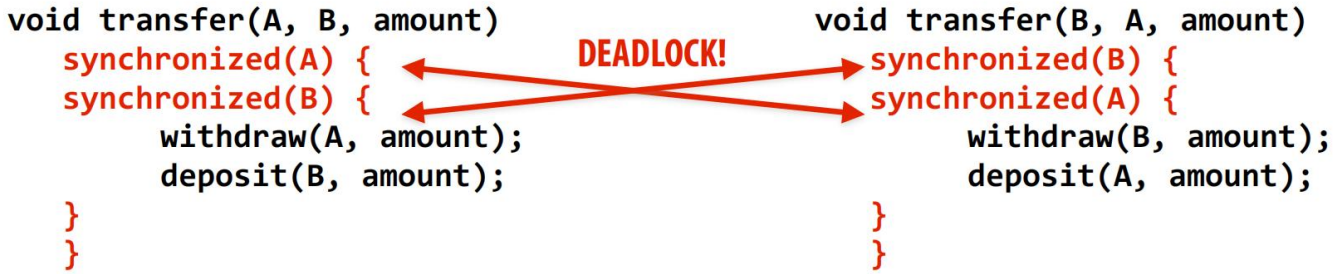
- Manually catch exceptions
  - Programmer provides undo code on a case by case basis
  - Complexity: what to undo and how…
  - Some side-effects may become visible to other threads
    - E.g., an uncaught case can deadlock the system

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



- Composing lock-based code can be tricky
  - Requires system-wide policies to get correct
  - Breaks software modularity
- Between an extra lock & 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);
    }
```

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

- 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 for transfer(A, B, $100) & transfer(B, A, $200)
  - Concurrency for transfer(A, B, $100) & transfer(C, D, $200)

Advantages of TM

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

Atomic() ≠ lock()+unlock()

- The difference
  - Atomic: high-level declaration of atomicity
  - Does not specify implementation/blocking behavior
  - Does not provide a consistency model
  - Lock: low-level blocking primitive
  - Does not provide atomicity or isolation on its own
- Keep in mind
  - Locks can be used to implement atomic(), 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., atomic sequence incorrectly split into two atomic blocks

lock vs atomic

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

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

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

- What is the problem with replacing synchronized with atomic?

Example: atomicity violation

- Programmer mistake: logically atomic code sequence separated into two atomic() blocks.

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

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

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

TM: summary + benefits

- TM = declarative synchronization
  - User specifies requirement (atomicity & isolation)
  - System implements in best possible way
- Motivation for TM
  - Difficult for user to get explicit sync right
    - Correctness vs. performance vs. complexity
  - Explicit sync is difficult to scale
    - Locking scheme for 4 CPUs is not the best for 64
  - Difficult to do explicit sync with composable SW
    - Need a global locking strategy
  - Other advantages: fault atomicity, …
- Productivity argument: system support for transactions can achieve 90% of the benefit of programming with fined_x0002_grained locks, with 10% of the development time.

Implementing TM

TM implementation basics

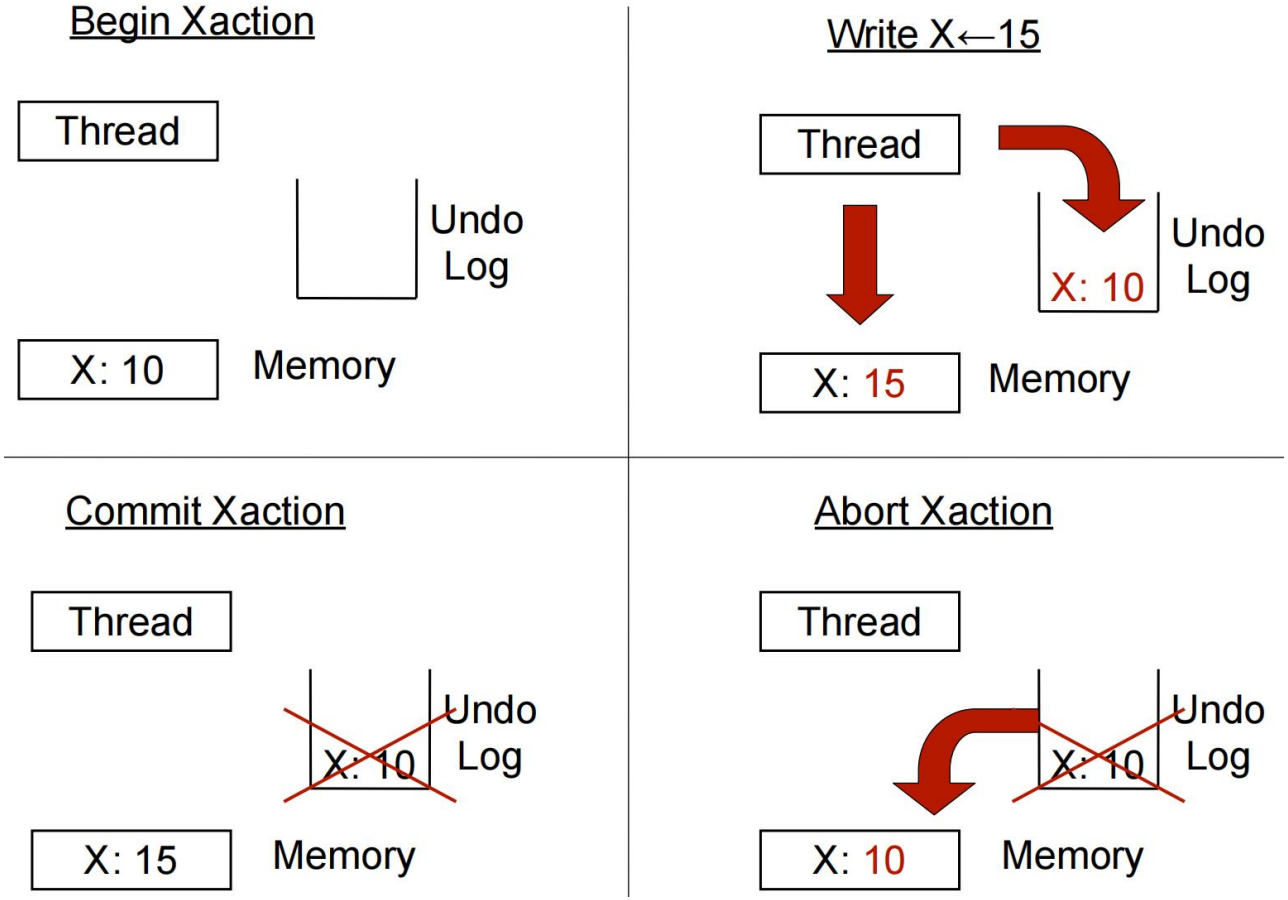
- TM systems must provide atomicity and isolation
  - Without sacrificing concurrency
- Basic implementation requirements
  - Data versioning (ALLOWS abort)
  - Conflict detection & resolution (WHEN to abort)
- Implementation options
  - Hardware transactional memory (HTM)
  - Software transactional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

Data versioning

- Manage uncommited (new) and commited (old) versions of data for concurrent transactions
  - Eager versioning (undo-log based)
  - Lazy versioning (write-buffer based)

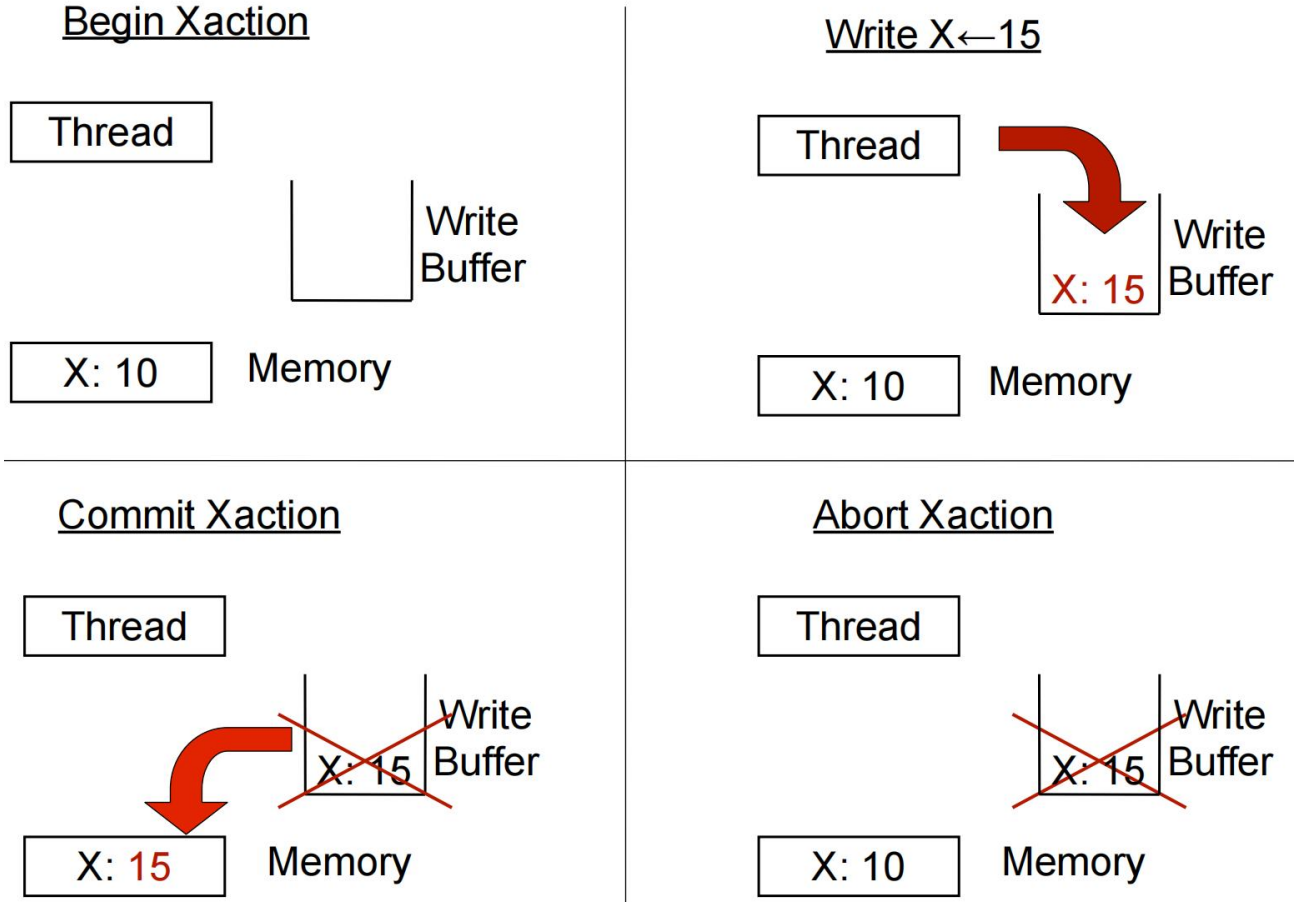
Eager versioning

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



Lazy versioning

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



Data versioning

- Manage uncommited (new) and commited (old) versions of data for concurrent transactions
- Eager versioning (undo-log based)
  - Update memory location directly
  - Maintain undo info in a log (per store penalty)
  - + Faster commit
  - – Slower aborts, fault tolerance issues (crash in middle of trans)
- Lazy versioning (write-buffer based)
  - Buffer data until commit in a write-buffer
  - Update actual memory location on commit
  - + Faster abort, no fault tolerance issues
  - – Slower commits

Conflict detection

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

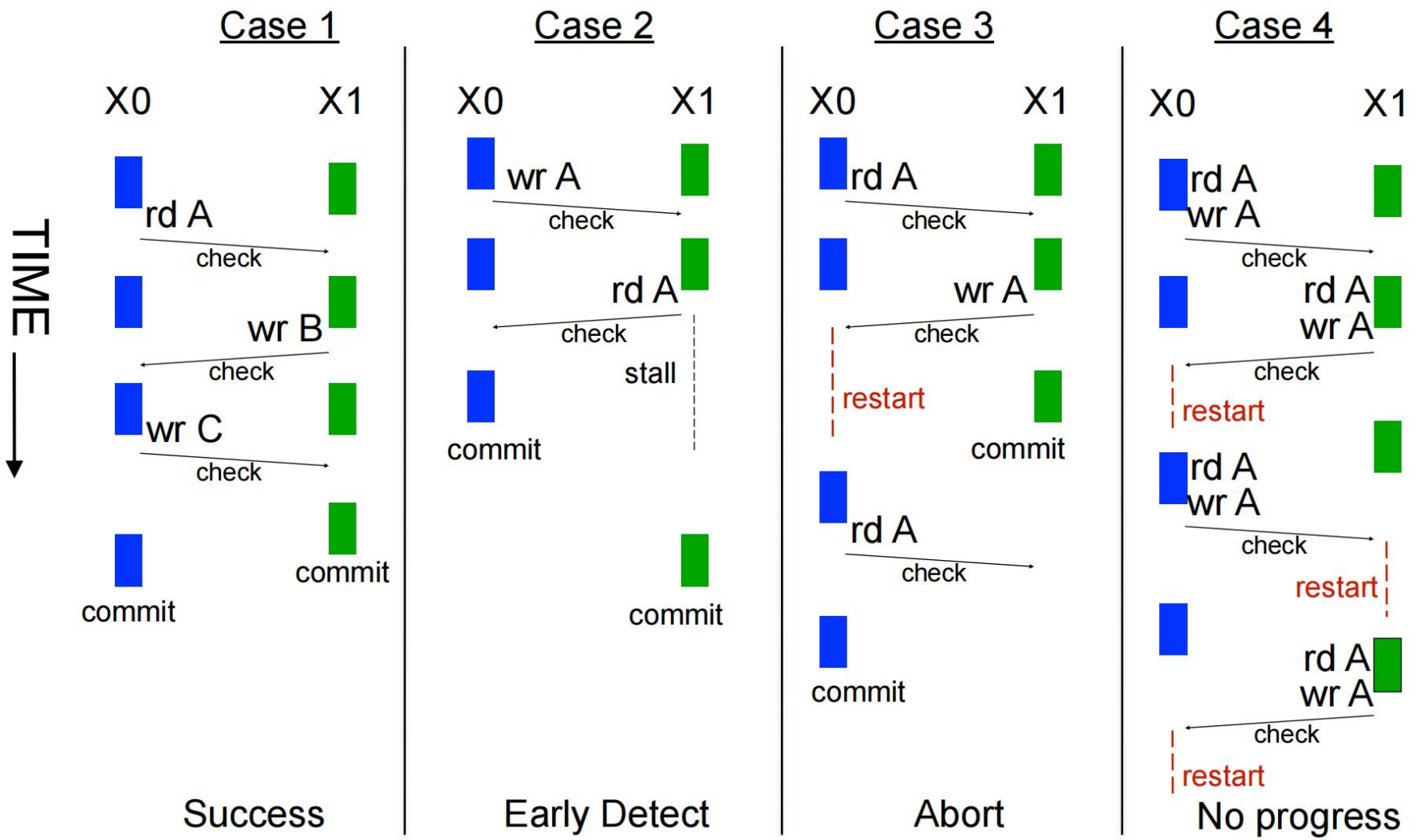
- Must track the transaction's read-set and write-set
  - Read-set: addresses read within the transaction
  - Write-set: addresses written within transaction

Pessimistic detection

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

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

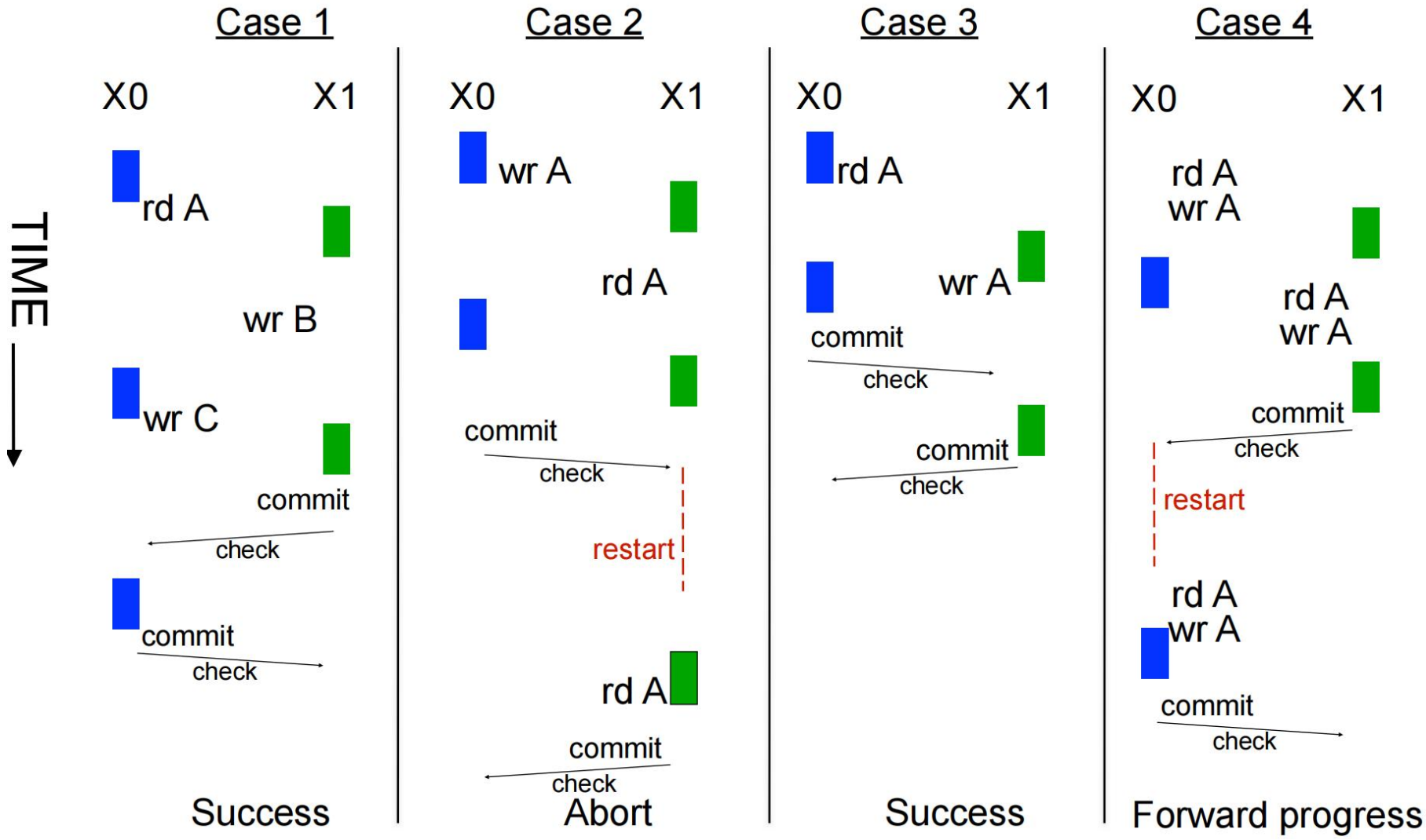
Pessimistic detection example



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 & pessimistic schemes together
  - Several STM systems use optimistic for reads and pessimistic for writes

Optimistic detection example



Conflict detection trade-offs

- Pessimistic conflict detection (a.k.a. “encounter” or “eager”)
  - + Detect conflicts early
    - Undo less work, turn some aborts to stalls
  - – No forward progress guarantees, more aborts in some cases
  - – Fine-grain communication
  - – On critical path

- Optimistic conflict detection (a.k.a. “commit” or “lazy”)
  - + Forward progress guarantees
  - + Potentially less conflicts, bulk communication
  - – Detects conflicts late, can still have fairness problems

Conflict detection granularity

- Object granularity (SW-based techniques)
  - + Reduced overhead (time/space)
  - + Close to programmer’ s reasoning
  - – False sharing on large objects (e.g. arrays)
- Word granularity
  - + Minimize false sharing
  - – Increased overhead (time/space)
- Cache line granularity
  - + Compromise between object & word
- Mix & match → best of both words
  - Word-level for arrays, object-level for other data, …

TM implementation space

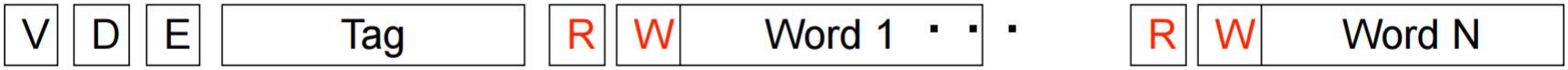
- Hardware TM systems
  - Lazy + optimistic: Stanford TCC
  - Lazy + pessimistic: MIT LTM, Intel VTM
  - Eager + pessimistic: Wisconsin LogTM
  - Eager + optimistic: not practical
- Software TM systems
  - Lazy + optimistic (rd/wr): Sun TL2
  - Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
  - Eager + 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 TM (HTM)

- Data versioning in caches
  - Cache the write-buffer or the undo-log
  - New cache meta-data 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 & directory coherence
- Notes
  - Register checkpoint must be taken at transaction begin

HTM Design

- Cache lines annotated to track read-set & 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

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